

“KOTLINLEARNING”

Elisa Pace - 1084405
Luca Marziliano - 1083813
Viorel Saran - 1085591

Progetto didattico per il corso di Programmazione Mobile 2020/2021

RELAZIONE PROGETTO: “KOTLIN LEARNING”

1 INTRODUZIONE

Il nostro progetto “Kotlin Learning” è una app definita per l’apprendimento del linguaggio di programmazione Kotlin, in cui si mira a dare all’utente una formazione base per l’acquisizione di nozioni tecniche al fine di poter comprendere e scrivere codice in linguaggio Kotlin.

La nostra applicazione è stata sviluppata con un’idea di “Learn and Test”, ciò significa che l’app mette a disposizione dell’utente una serie di insegnamenti teorici, con la possibilità di testare quello appena appreso tramite dei quiz.

2 SVILUPPO ANDROID

2.1 REQUISITI

I requisiti che ci siamo posti, nello sviluppo della nostra app, comprendono:

- La presenza di un test delle conoscenze, per dare all’utente la possibilità di conoscere il suo stato di preparazione sull’argomento; questo dovrà essere svolto una sola volta al primo lancio dell’app ed in base al punteggio ottenuto, potrà sbloccare l’accesso a sezioni diversi dell’app.
- La definizione di una linea guida che l’utente dovrà rispettare per proseguire nell’app; perciò, sarà possibile accedere ad un argomento, solo nel momento in cui si è superato con esito positivo le domande fornite nell’argomento precedente.
- La possibilità di attingere in qualsiasi momento alle nozioni teoriche presenti
- La possibilità di effettuare quiz di natura differente, per verificare le nozioni apprese, ogni volta che l’utente desidera
- La capacità di registrare il punteggio migliore totalizzato dall’utente, sui quiz di ogni argomento

Inoltre, nello svolgimento dei quiz, si è deciso di NON rivelare la risposta corretta alle domande fornite, per non facilitare troppo i test e per permettere all’utente di arrivare alla soluzione autonomamente.

2.2 CASI D’USO

Nella figura 1, vengono illustrati i casi d’usi, nello specifico viene descritta la possibilità che l’utente ha di: effettuare un test per valutare le sue conoscenze, visionare la teoria relativa ad ogni argomento, eseguire quiz per verificare le informazioni apprese e visualizzare i progressi e i punteggi effettuati.



Figura 1-Casi d'uso

2.3 ARCHITETTURA DELL'APP

L'architettura con cui è stata strutturata l'applicazione, segue il pattern MVVM, dove l'acronimo sta ad indicare rispettivamente **Model**, **View** e **ViewModel**. In più si è deciso di utilizzare un altro componente, non obbligatorio, ma fortemente consigliato, ovvero le **Repository**, che svolgono la funzione di tramite, tra il model e il viewmodel; come riporta la figura 2.

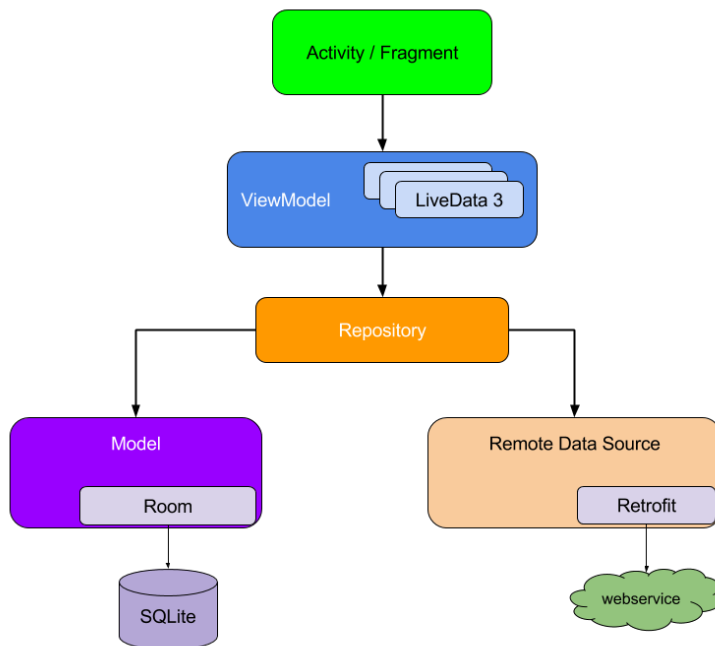


Figura 2-MVVM pattern

2.3.1 Model

Il **Model** è rappresentato da quelle classi che contengono i dati dell'applicazione e che hanno il compito di salvare permanentemente, modificare, inserire e cancellare le informazioni relativa ad essa.

2.3.1.1 Progettazione della componente dati

Nel nostro caso si è deciso di interfacciarsi con un database locale, SQLite, messo a disposizione da Android, che poi verrà affiancato dalla libreria Room. Questo permette di avere un livello di astrazione superiore a SQLite, ma sempre sfruttando l'espressività del linguaggio di query; inoltre funge anche da libreria per "Object Relational Mapping", permettendo di convertire dati dalla loro rappresentazione in database in oggetti manipolabili all'interno del codice e viceversa.

Diagramma E-R

Il seguente diagramma Entità-Relazione (figura 3) funge da rappresentazione grafica per la progettazione di una base di dati, che verrà utilizzata per il funzionamento ed il rispetto dei requisiti posti per applicazione.

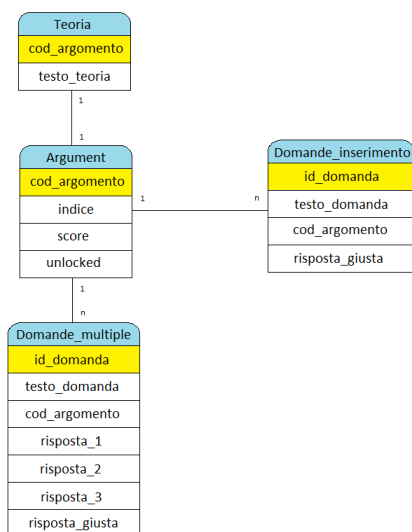


Figura 3- Diagramma E-R

L'applicazione dovrà essenzialmente memorizzare i dati relativi agli argomenti trattati e per una agevolazione nelle operazioni di aggiornamento/inserimento, si è deciso di suddividere gli aspetti più caratterizzanti in diverse entità; perciò, avremmo quattro entità identificate dai seguenti nomi: "Argument", "Teoria", "Domande Multiple", "Domande Inserimento".

L'entità "Argument" è composta da quattro attributi, con identificatore "cod_argomento", che identifica il nome dell'argomento, e tre descrittori: "indice", numero assegnato ad ogni argomento in base all'ordine di presentazione, "score", il punteggio effettuato dall'utente, e "unlocked", contiene un valore 1 o 0 e specifica se l'argomento è stato sbloccato; inoltre è l'entità che viene aggiornata costantemente.

L'entità "Teoria" è caratterizzata da due attributi, un identificatore "cod-argomento", che identifica il nome dell'argomento e un descrittore "testo_teorìa", che descrive tutta la teoria affrontata in quell'argomento specifico e collegato all'entità "Argument" da una relazione (1,1), in quanto ogni argomento ha la sua teoria e ogni teoria tratta di un specifico argomento.

L'entità "Domande_Multiple" è costituita da sette attributi, di cui un identificatore "id_domanda" e da alcuni attributi, come: "testo_domanda", che contiene il testo della domanda, "cod_argomento" e le varie risposte possibili, compresa quella giusta, della domanda. La relazione che sussiste tra l'entità "Argument" e "Domande Multiple" è di (1,n), dato che un argomento può avere più domande, ma una domanda è riferita ad un singolo argomento.

L'entità "Domande Inserimento", è quasi composta dagli stessi attributi dell'entità precedente, ma trattando tipologie di domande differenti avrà un'unica risposta, ovvero quella giusta.

Il tutto viene riassunto nella figura 4 che descrive il dizionario delle entità.

NOME ENTITA'	DESCRIZIONE	ATTRIBUTI	IDENTIFICATORE
Argument	Descrive la parte dinamica di ogni argomento e che registra i progressi dell'utente	Codice argomento(stringa), Indice(numerico), Score(numerico), Unlocked(numerico)	Codice argomento(stringa)
Teoria	Raccoglie le nozioni teoriche di ogni argomento	Codice argomento(stringa), Testo domanda(stringa)	Codice argomento(stringa)
Domande Multiple	Contiene tutte le domande che hanno risposta multipla	ID Domanda(numerico), Testo Domanda(stringa), Codice argomento(stringa), Risposta 1(stringa), Risposta 2(stringa), Risposta 3(stringa), Risposta Giusta(stringa)	ID Domanda(numerico)
Domande Inserimento	Contiene tutte le domande che hanno una risposta, che deve essere inserita da tastiera	ID Domanda(numerico), Testo Domanda(stringa), Codice argomento(stringa), Risposta Giusta(stringa)	ID Domanda(numerico)

Figura 4-Dizionario Entità

Modello logico

Nello schema seguente (figura 5), gli attributi in grassetto e sottolineati indicano la chiave primaria per ogni entità; mentre gli attributi in corsivo presentano un vincolo di integrità referenziale.

Argument (cod_argomento, indice, score, unlocked)
 Teoria (cod_argomento, testo_teorica)
 DomandeMultiple (id_domanda, testo_domanda, cod_argomento, risposta_1, risposta_2, risposta_3, risposta_giusta)
 Domande Inserimento ((id_domanda, testo_domanda, cod_argomento, risposta_giusta)

Figura 5-Modello Logico

2.3.1.2 Componenti principali di Room

Di seguito verrà mostrato come è stata gestita la progettazione dei dati e come sono stati sistemati in accordo con l'utilizzo della libreria Room, schematizzato in figura 5.

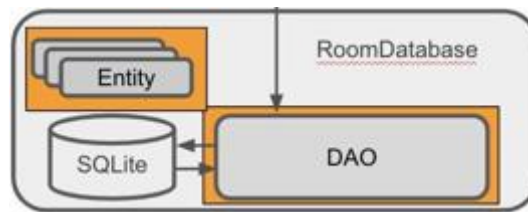


Figura 6 - Componenti Room

Entity

È una classe che viene mappata ad una tabella di un database SQLite ed utilizza diverse annotazioni come: **@Entity** e **@PrimaryKey** rispettivamente per indicare il nome della tabella (nel caso in cui si utilizzi un database già precompilato, il nome dato alla tabella deve corrispondere con quella già esistente nel database inserito) e indicare quale tra gli attributi presenti è l'identificatore. Per effettuare "Object Relational Mapping" viene utilizzata una data class, una particolare classe che viene utilizzata per memorizzare un insieme di dati. Vedi il pezzo di codice in esempio (figura 7)

```

6
7  @Parcelize
8  @Entity(tableName = "domande_multiple")
9  data class DomandeMultiple (
10
11      @PrimaryKey(autoGenerate = true)
12      val id_domanda: Int,
13      val testo_domanda: String,
14      val cod_argomento: String,
15      val risposta_1: String,
16      val risposta_2: String,
17      val risposta_3: String,
18      val risposta_giusta: String,
19
20
21  ):Parcelable
  
```

Figura 7- Entity DomandeMultiple

Dao

L'acronimo sta per Data access object, sono delle classi astratte o interfacce e gestiscono l'accesso al database. Nel nostro caso si può notare, dalla figura 8, che alcune chiamate query ritornano gli stessi valori, ma ciò che cambia è il modo in cui vengono gestite. Va specificato che generalmente Room gestisce le query in maniera sincrona, mentre nel caso di `getAllArgument()` gli elementi restituiti sono contenuti all'interno di un LiveData (che gestisce i task in modo asincrono), un componente lifecycle-aware, che osserva e notifica immediatamente ogni cambiamento agli osservatori, invece nel caso di `getAllArgumentwithCouroutine()` vengono usate le coroutine; perciò, con la keyword "suspend" identifico una funzione che non sia bloccante, quindi verrà sospesa finché la funzione (che girerà su un altro thread) non sarà conclusa, eseguendo un resume.

```
@Dao
interface ArgumentDao {

    @Query(value: "SELECT * FROM argument ORDER BY indice ASC")
    fun getAllArgument(): LiveData<List<Argument>>

    @Query(value: "SELECT * FROM argument ORDER BY indice ASC")
    suspend fun getAllArgumentwithCouroutine(): List<Argument>
}
```

Figura 8- Dao ArgumentDao

Database

I dati dell'applicazione sono salvati all'interno di un database locale precompilato, che si compone di quattro tabelle: una contenente i valori degli argomenti, e viene modificata per salvare i progressi dell'utente, un'altra invece relativa alla teoria, un'altra ancora alle domande con risposta multipla ed infine l'ultima relativa alle domande con risposta inserita da tastiera. Queste ultime tre sono tabelle che non vengono mai modificate durante il funzionamento dell'app. In figura 9 viene rappresentato l'istanza iniziale della tabella Argument.

cod_argomento	indice	score	unlocked
Filtro	Filtro	Filtro	Filtro
Variabili	1	0	1
Stringhe	2	0	0
Condizioni e Cicli	3	0	0
Funzioni	4	0	0
Null-Safety	5	0	0
Array e Collection	6	0	0
Classi	7	0	0
Ereditarietà	8	0	0
Lambda Functions	9	0	0

Figura 9- tabella Argument

Nel caso di un database Room, creiamo una classe con la seguente annotazione `@Database` e ci includiamo una lista di entity, poi definiamo una classe astratta che estende `RoomDatabase()` ed

infine definiamo dei metodi astratti che ritornano ognuno un relativo dao, questo va fatto per ogni tabella che si è definita. Da notare che per importare un database precompilato è necessario: settare exportSchema a true ed inserire dopo Room.databaseBuilder().createFromAsset() definendo dentro le parentesi il percorso in cui si è inserito il database, figura 10.

```
@Database(entities = [Argument::class, Teoria::class, DomandeInserimento::class, DomandeMultiple::class], version = 1, exportSchema = true)
abstract class AppDatabase:RoomDatabase() {

    abstract fun argumentDao():ArgumentDao
    abstract fun teoriaDao():TeoriaDao
    abstract fun domandemultipleDao():DomandeMultipleDao
    abstract fun domandeInserimentoDao():DomandeInserimentoDao

    companion object{
        @Volatile
        private var INSTANCE: AppDatabase? = null
        fun getInstance(context: Context): AppDatabase { return INSTANCE ?: synchronized(lock: this) {
            INSTANCE ?: Room.databaseBuilder( context.applicationContext, AppDatabase::class.java, name: "kotlin_learning_database"
            )

            //allowMainThreadQueries()
            .createFromAsset( databaseFilePath: "database/argument.db")
            .build()
            .also { INSTANCE = it }

        }
    }
}
```

Figura 10-AppDatabase

2.3.2 Repository

E' una class astratta che gestisce operazioni tra differenti fonti di dati e svolge una funzione di mediatore tra il database ed il viewmodel. Nel nostro caso viene utilizzato per svolgere con maggiore agevolezza operazioni che vanno svolte su thread differenti dal main(nel caso delle coroutine, figura 11) e per avere una miglior suddivisione del codice.

```
class ArgumentRepository(private val argumentDao:ArgumentDao) {
    fun readallArgument():LiveData<List<Argument>> = argumentDao.getAllArgument()

    fun getArgument(argumento:String):LiveData<Argument>{...}
    suspend fun getAllArgumentwithCoroutine(): List<Argument>{
        val argomenti:List<Argument>
        withContext(Dispatchers.IO){ this: CoroutineScope
            argomenti= argumentDao.getAllArgumentwithCouroutine()
        }
        Log.i( tag: "ArgumentRepository", msg: "Dentro funzione getAllArgumentwithCoroutine in repository"
        return argomenti
    }
}
```

Figura 11-ArgumentRepository

2.3.3 ViewModel

Sono delle classi che vengono associate a delle componenti view(activity, fragment) e sono legate al loro ciclo di vita, questo permette di mantenere i dati , nonostante improvvisi cambiamenti di configurazione e contengono esclusivamente la parte logica di gestione delle view.

Il ViewModel non può contenere riferimenti al layout di un fragment o activity, ma a volte è necessario aggiornare delle view con dati contenuti nel viewModel, per questo si possono definire variabili di layout, che una volta agganciate alla reale istanza di viewModel, utilizzando il DataBinding, facilitano notevolmente queste operazioni di aggiornamento.

```
<data>
    <variable
        name="domademultiple"
        type="com.example.kotlinlearning.viewmodel.QuizBottoneViewModel" />
</data>
<TextView
    android:id="@+id/testo_domanda_multipla"
    android:layout_width="300dp"
    android:layout_height="230dp"
    android:layout_marginTop="45dp"
    android:background="@drawable/contorno"
    android:gravity="center"
    android:paddingHorizontal="10dp"
    android:text="@{domademultiple.domandaAttuale.testo_domanda}"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Figura 12-Variabili di layout in `qui_bottone.xml`

Nel nostro caso ad ogni fragment è stato associato un relativo viewModel

2.3.4 View

Le View sono classi che gestiscono la parte UI di un layout ad esse associate, come fragment e activity. Nella applicazione utilizziamo tre Activity, WelcomeScreen, MainActivity e MainActivity2 ed a due di queste (MainActivity e MainActivity2) associamo un navigation graph ognuna. Il primo ad essere utilizzato è il `nav_graph_2` associato al MainActivity2 e viene usato per svolgere la parte di app relativa al test delle conoscenze, invece il secondo `nav_graph` (figura 13) è associato al MainActivity e coinvolge tutto il resto del funzionamento dell'app.

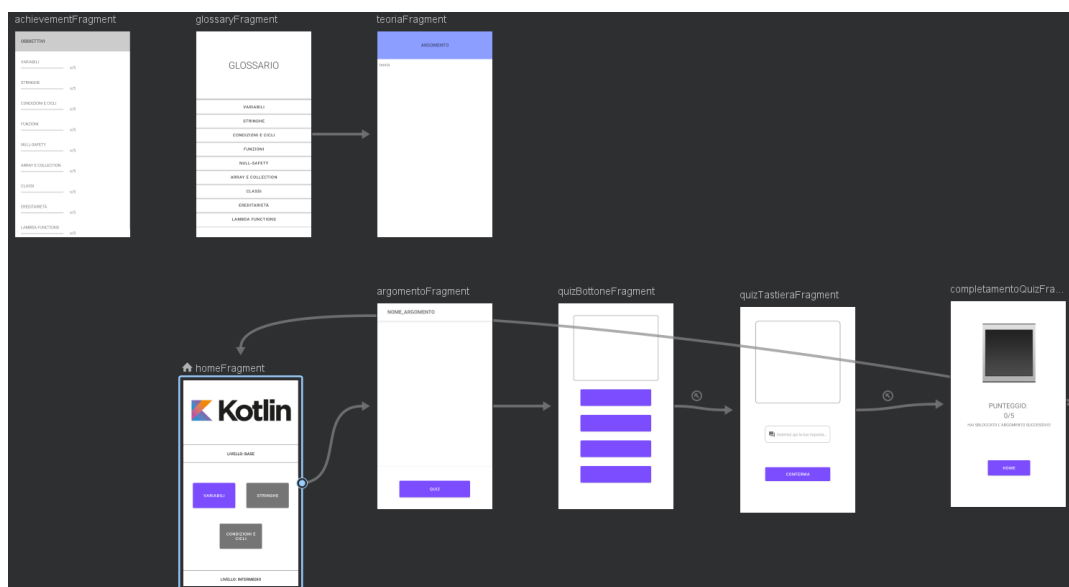


Figura 13- `nav_graph`

2.4 USER INTERFACE

2.4.1 Mappa app

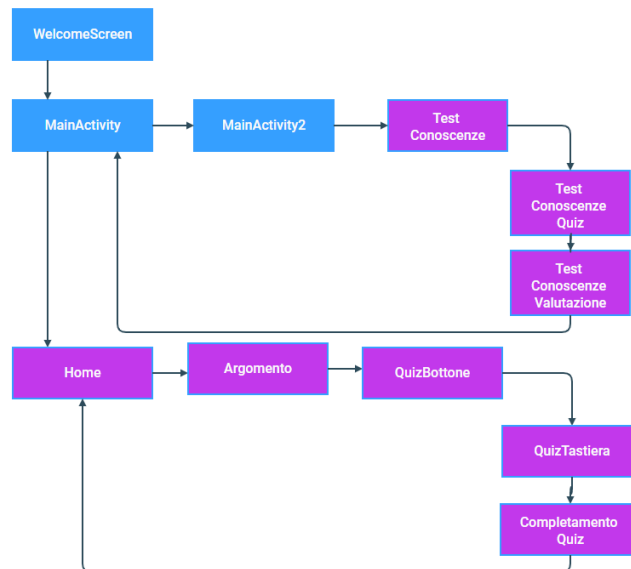


Figura 14- Mappa dell'app, in blue le activity, in fucsia i fragment

2.4.2 Mockup

Appena l'utente avvia l'app parte una schermata che raffigura il logo dell'app.



Figura 15- WelcomeScreen

La prima volta che l'utente lancerà l'app gli verrà proposta questa schermata,



Figura 16-Test Conoscenze

se l'utente preme il pulsante "procedi" allora si presenterà la schermata seguente ed



Figura 17- Quiz Test Conoscenze

una volta completate tutte le domande, verrà fornita una schermata con i risultati ottenuti.



Figura 18-Test Conoscenze Valutazioni

In base ai risultati ottenuti si avrà dei livelli sbloccati differenti, ma nel caso di un utente che ha optato per saltare il test delle conoscenze o, che abbia totalizzato un punteggio non sufficiente a sbloccare una sezione (Discreto, come in questo caso), la sua app avrà questo aspetto:

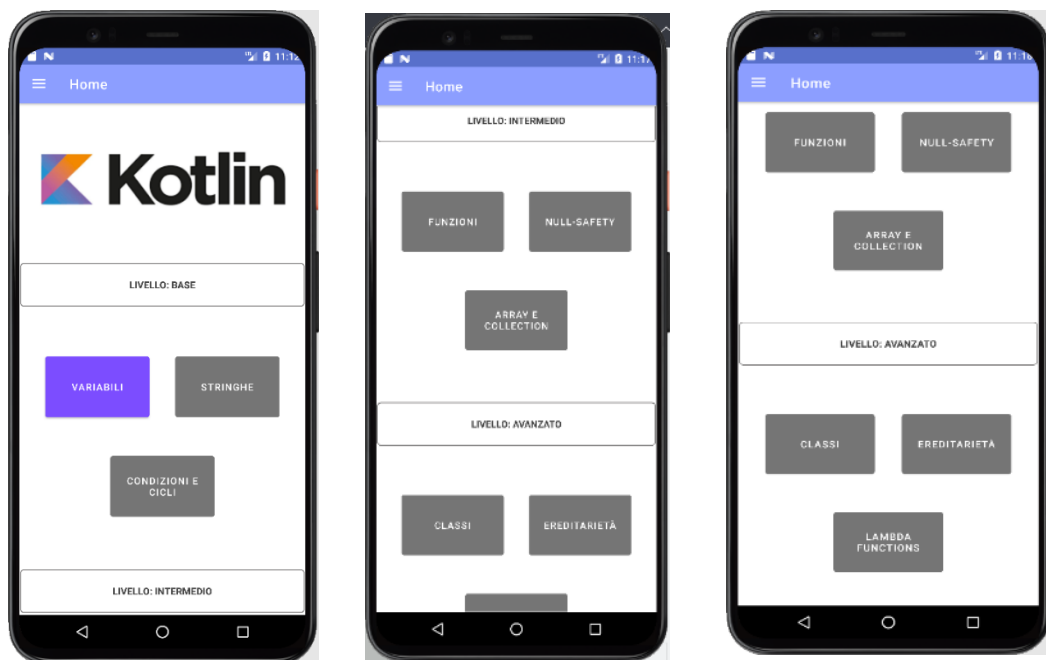


Figura 19-Home

Le sezioni in cui abbiamo deciso di dividere il progetto in base alla difficoltà sono:

- Variabili
- Stringhe
- Condizioni e Cicli
- Funzioni
- Null-Safety
- Array e Collection
- Classi
- Ereditarietà
- Lambda Functions

Dato che per il momento “Variabili” è l’unico argomento accessibile, accedendoci l’utente avrà la possibilità di leggere la teoria associata e di eseguire un quiz per determinare la sua conoscenza nell’argomento trattato.

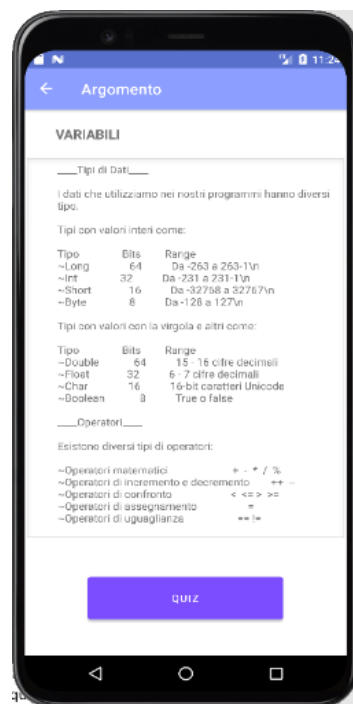


Figura 20-Argomento scelto

Accedendo al bottone del quiz, l’utente verrà sottoposto a una sequenza standard di esercizi, quattro domande a risposta multipla e per poi finire una domanda aperta con inserimento da tastiera.

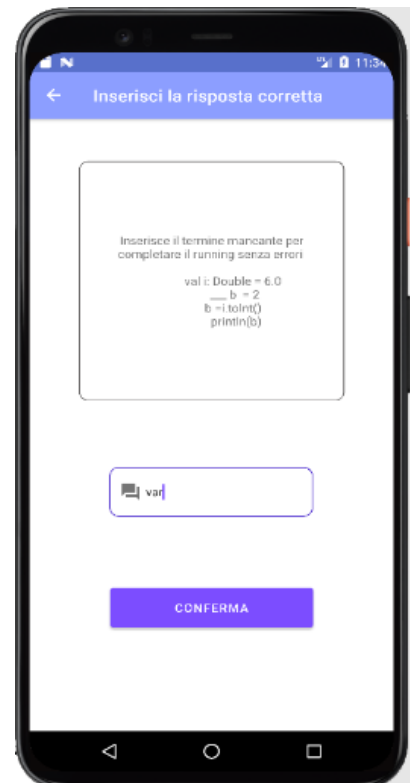


Figura 21-Quiz scelta multipla e qui con risposta inserita da tastiera

Al completamento del quiz all'utente viene poi presentato il punteggio totalizzato nel quiz.



Figura 22- Completamento Quiz, esito positivo e esito negativo

Se l'utente ha superato, con un punteggio maggiore o uguale alla metà delle domande proposte, il test , allora può accedere all'argomento successivo, sbloccando il bottone nella schermata Home.

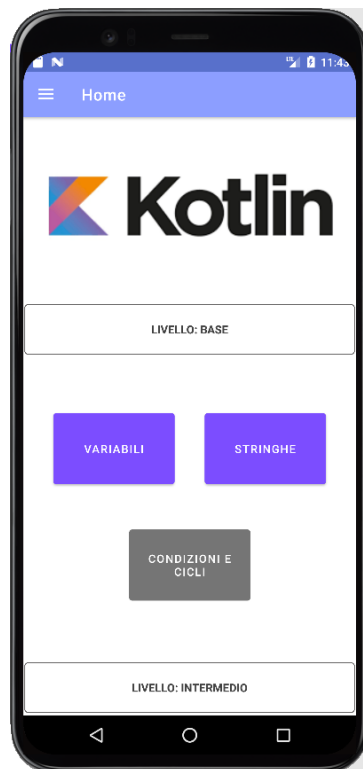


Figura 23- Home nuovo argomento sbloccato

Per permettere all'utente di vedere i punteggi che ha totalizzato in ciascun argomento, abbiamo implementato nel navigation drawer un collegamento alla schermata obiettivi, dove i punteggi sono visibili tramite una rappresentazione con barra a completamento e una textview con il punteggio totalizzato sul massimo delle domande di quell'argomento.

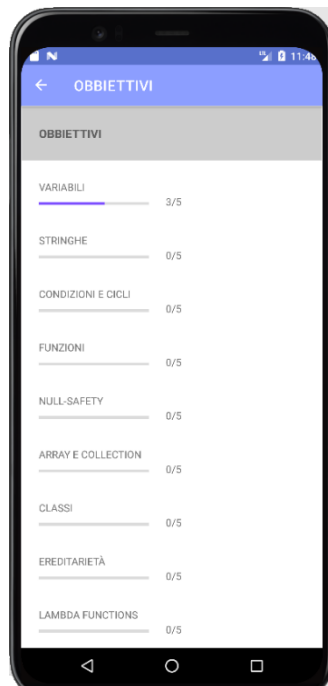


Figura 24- obiettivi

Inoltre, è stato implementato nel navigation drawer, anche la voce glossario per accedere alle varie spiegazioni degli argomenti, in cui cliccando sul bottone dell'argomento interessato si viene indirizzati in una schermata in cui viene mostrata la teoria rilevante ad esso.

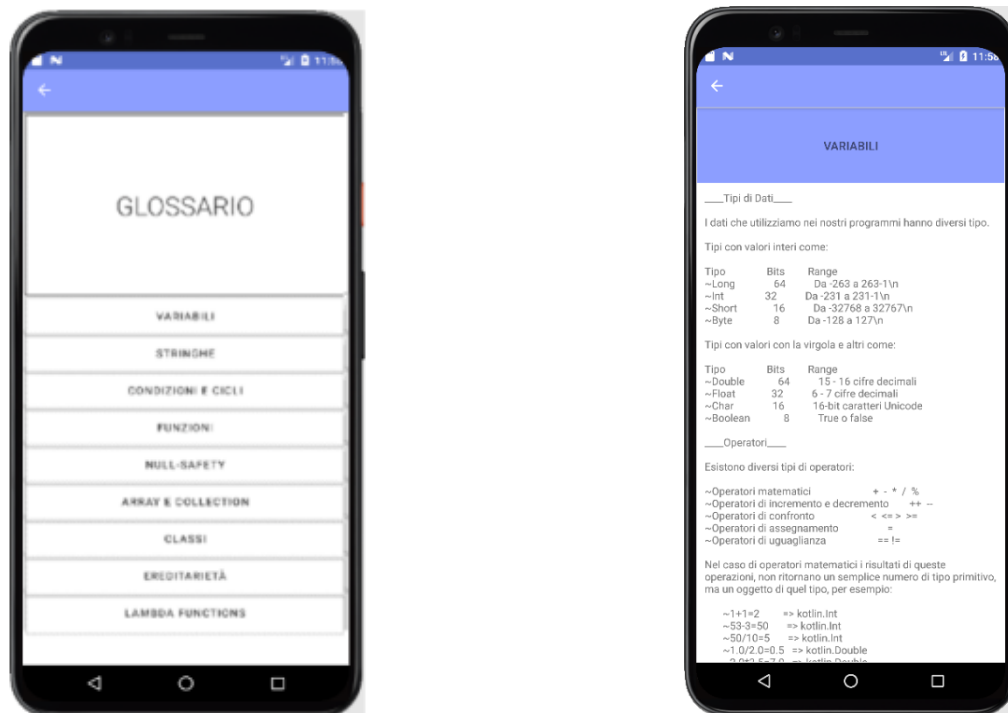


Figura 25-Glossario

2.5 SVILUPPO

2.5.1 Funzionamento generale

Tra i requisiti che ci siamo posti, compariva la necessità di proporre un test delle conoscenze, solo nel primo utilizzo dell'applicazione, per questo è sorta la necessità di separare in due parti ben distinte la gestione del pre-test, dal resto dell'applicazione; perciò, si è deciso di utilizzare un activity ,MainActivity2, a cui associare il nav_graph_2, contenente tre fragment, rispettivamente TestConoscenzeFragment, QuizTestConoscenzeFragment, TestConoscenzeValutazioneFragment, e un'altra activity, MainActivity, a cui legare un altro nav_graph, questa volta contenente tutti i fragment relativi al resto del funzionamento dell'app. Va precisato che all'interno del nav_graph, sono presenti anche tre fragment, rispettivamente Glossary, Achievement e TeoriaFragment, collegati all'interno del navigation drawer e distinti dei rimanenti fragment. Dopo aver separato le due diverse funzionalità dell'app, in due parti differenti, abbiamo trattato il problema di dover effettuare il test delle conoscenze una sola volta, per questo si è pensato di utilizzare un Share Preference che contenesse un valore boolean; nel caso in cui il valore contenuto al suo interno fosse stato true, allora era la prima volta che l'utente utilizzava l'applicazione, mentre nel caso contrario l'utente aveva già utilizzato più di una volta l'app. Ci siamo serviti poi di questo valore all'interno dell'activity, MainActivity, confrontandolo in un if,: in caso di valore uguale a true, modificava il suo valore in false e poi eseguiva un intent al MainActivity2, in caso di false, veniva invocato il primo fragment del nav_graph, HomeFragment. In questo modo si impedisce di utilizzare più di una volta la funzione pre-test. Il resto dell'applicazione si svolge eseguendo cicli dei seguenti fragment: HomeFragment, ArgomentoFragment, QuizBottoneFragment, QuiTastieraFragment e CompletamentoQuizFragment.

2.5.2 Funzionamento delle specifiche componenti

TestConoscenze

Nel relativo Fragment sono presenti due setonlicklistener associati a due bottoni, rispettivamente Salta e Procedi. Al verificarsi dell'evento della premuta del bottone Salta, viene eseguito un intent al MainActivity, mentre nel caso della premuta del bottone procedi, viene eseguito una navigazione tra fragment, da TestConoscenze a QuizTestConoscenze con il passaggio di un safe args contenente le domande da eseguire nel fragment successivo. Per ottenere le domande da svolgere nel test successivo, viene effettuata una chiamata al database, tramite la funzione suspend `getMultipleQuestion()`, che verrà eseguita nel thread IO e al termine della sua esecuzione aggiornerà la variabile `mutablelist<DomandeMultiple>` `allMultiQuestion` con una lista di `DomandeMultiple` presenti nel database. Questa lista verrà poi utilizzata nella funzione `selectQuestionfromArgument`, insieme alla lista `nameargument` per ottenere una lista contenente due domande per ogni argomento trattato.

```
suspend fun getMultipleQuestion(){
    allMultiQuestion=repository.getMultipleQuestion().toMutableList()
    Log.i(tag: "TestConoscenzeViewModel", msg: "La lista di domandemultiple è stata estratta con successo")
}

//ho necessita di inizializzare la variabile globale allMultiQuestion, ma getMultipleQuestion() è una funzione suspend e
// quindi uso un viewModelScope per usare la funzione
fun domandeMulti(){
    viewModelScope.launch{ this: CoroutineScope
        getMultipleQuestion()
    }
}
```

```
class DomandeMultipleRepository(private val domandemultipleDao: DomandeMultipleDao) {

    suspend fun getMultipleQuestion():List<DomandeMultiple>{
        val domandemultiple:List<DomandeMultiple>

        withContext(Dispatchers.IO) { this: CoroutineScope
            domandemultiple = domandemultipleDao.getAllQuestion()
        }
        Log.i(tag: "DomandeMultiRepository", msg: "Dentro metodo getMultipleQuestion in repository,

        return domandemultiple
    }
}
```

Figura 26-chiamata al database con utilizzo di coroutine, `withContext(Disptachers.IO)` e `viewModelScope.launch{}`

QuizTestConoscenze

Questo fragment ha il compito di capire quale bottone ha premuto l'utente, verificare se il bottone premuto contenga la risposta corretta e verifica che il numero di quiz eseguiti, sia minore del numero dei quiz totali; se questa condizione si verifica, allora viene fatto il refresh del layout (`binding.invalidateAll`) caricando la domanda e le risposte successive e aggiornando la progressbar, mentre nel caso in cui la condizione non si verifica, viene eseguita una navigazione tra fragments da QuizTestConoscenze a TestConoscenzeValutazione. Tutte queste operazioni vengono svolte all'interno di un oggetto listener, che sovrascrive la funzione on click e che poi verrà associato a tutti

i bottoni presente nel layout. Per verificare che l'utente abbia premuto il bottone contenente la risposta corretta, viene presa la descrizione di quel bottone, con un `getContentDescription`, a questa descrizione verrà associato un valore intero, che può assumere valori da zero a tre, che poi verrà passato come parametro alla funzione `correctAnswer`, che a sua volta incrementerà la variabile `nriscorrette`, se la risposta coincide con la risposta giusta della domanda corrente e incrementerà la variabile `indiceDomande`, utilizzata per prendere la domanda successiva. Nella navigazione al fragment successivo, viene passato un `safe args` contenente la variabile `nriscorrette`. Nel `viewmodel`, invece, vengono estese due interfacce: la prima `GestioneDomande` che contiene alcuni metodi ricorrenti per trattare le domande e la seconda `NumeroDomande`, che viene utilizzata per salvare il numero delle domande utilizzate sia nei test delle conoscenze che nei quiz.

TestConoscenzeValutazione

In questo fragment vengono eseguite delle operazioni per mostrare a display delle componenti differenti in base al risultato ottenuto dall'utente e quindi in base al numero di risposte corrette ottenute tramite `safe args`. Una volta controllati i risultati, vengono eseguiti degli inserimenti, con `replace`, al database ed il numero di elementi inseriti varierà in base al numero delle risposte corrette effettuate. La funzione che controlla e aggiorna i risultati, prende il nome di `checkandUpdateResult`; questa funzione prenderà come parametro le risposte corrette e in base ad esse, aggiornerà una variabile intera `result`, che assume valori da zero a tre, che verrà restituita da questa funzione; in più sceglierà un numero passatogli di elementi, dalla lista `<Argument> listadiArgomentiIniziali`, che verranno modificati settando la proprietà `unlocked` uguale a uno e poi inseriti nel database attraverso la funzione `insertMoreArgument()`.

Infine, premendo il bottone home, il `setOnClickListener` ad esso associato eseguirà un intent alla `MainActivity`.

```
fun insertMoreArgument(vararg argomento: Argument) {
    viewModelScope.launch(Dispatchers.IO) { this: CoroutineScope
        repository.insertMoreArgument(*argomento)
    }
}
```

Figura 27-funzione che inserisce più oggetti di tipo Argument

Home

Il seguente fragment al compito di aggiornare layout, rendendo cliccabile i vari bottoni e cambiandone il colore; questa operazione viene fatta utilizzando un observer, che osserva un `LiveData` contenente una lista `<Argument> listOfArgument` presa dal database. Utilizzando un `livedata`, observer verrà attivato ogni volta che si verificherà un cambiamento all'interno del database, questo ci permette di aggiornare correttamente il nostro layout ogni qualvolta la proprietà `unlocked` di ogni oggetto `Argument` della lista `listOfArgument`, viene settato ad 1. Inoltre viene creato un oggetto listener, che verrà associato ad ogni bottone del layout ed eseguirà una navigazione al fragment successivo, `ArgomentoFragment`, passandogli con un `safe args`: la descrizione del bottone premuto, contenente il nome dell'argomento e una lista di oggetti teoria, che sono presi attraverso la chiamata `getTheory()` al database, nel `viewmodel`.

```

var listOfargument:LiveData<List<Argument>> = MutableLiveData<List<Argument>>()

init{
    val argomentoDao= AppDatabase.getInstance(application).argumentDao()
    repository=ArgumentRepository(argumentoDao)
    //inizializzo il livedata con i valori contenuti nel database
    listOfargument= repository.readallargument()

    fun getAllArgument(): LiveData<List<Argument>>{
        return listOfargument
    }
}

```

Figura 28-Nel viewmodel, chiamata al database con livedata

```

//setto un observer per aggiornare la lista "listargomenti" una volta che la query è stata eseguita
// e prende la lista aggiornata per esegue la funzione CheckView
homeviewmodel.getAllArgument().observe( viewLifecycleOwner, Observer { argument ->
    Log.i( tag: "HomeFragment", msg: "Il contenuto della lista argomenti è: ${argument.toString()}")

    CheckView(argument)
    Log.i( tag: "HomeFragment", msg: "Completato il settaggio del layout grafico")

})

```

Figura 29-Observer nel fragment

Argomento

Nel seguente fragment viene mostrata la teoria di ogni argomento, nello specifico viene modificato la TextView iniziale, con il valore contenente il nome dell'argomento passato dal safe args e viene visualizzato il contenuto della proprietà testo_domanda dell'oggetto Teoria corrispondente all'argomento passato. In più nel viewmodel viene eseguita una chiamata al database, che restituisce una lista<DomandeMultiple> contenute nel database, che poi verrà utilizzata dalla funzione selectQuestionfromArgument e restituirà una lista di domande multiple contenente le domande riguardante quel singolo argomento passato; questa lista di domande verrà poi passata al fragment successivo, QuizBottoneFragment, nel momento in cui l'utente premerà il bottone quiz. Quest'ultima operazione viene fatta perché l'estrazione da database richiede una certa quantità di tempo di esecuzione e queste informazioni devono essere immediatamente disponibile per essere illustrate nel layout; perciò, vengono eseguite in questo fragment e passate a quello successivo per fornire il tempo necessario di completare l'operazione senza problemi.

QuizBottone

In questo fragment vengono svolte quasi le stesse operazioni eseguite in QuizTestConoscenze, l'unica differenza si verifica nel fatto che all'interno del viewmodel viene eseguita una chiamata al database getInputQuestion che restituisce una lista di DomandeInserimento, che poi verranno passate al fragment successivo, QuizTastieraFragment, tramite safe args, insieme a l'argomento corrente e le risposte corrette effettuate dall'utente. L'operazione di chiamata al database getInputQuestion viene eseguita per la stessa motivazione spiegata sopra.

QuizTastiera

Anche in questo fragment le operazioni che vengono eseguite sono quasi le stesse ripetute in QuizBottone, la differenza si trova nel fatto che, l'utente deve inserire la risposta corretta all'interno dell' EditText utilizzando la tastiera; in questo caso la risposta sarà considerata corretta se la parola inserita dall'utente, corrisponde alla risposta giusta della domanda corrente. Nel caso in cui l'utente dovesse proseguire, premendo il pulsante conferma, senza inserire nessuna risposta, verrà visualizzato un ToastBar che invita l'utente ad inserire una risposta prima di proseguire. Nel passaggio al fragment successivo, CompletamentoQuiz, viene allegato un safe args contenente l'argomento corrente, il totale delle risposte corrette, e una lista che contiene l'istanza attuale della tabella Argument, che verrà utilizzata nel fragment successivo.

CompletamentoQuiz

Anche in questo caso le operazioni che vanno svolte sono simili a quelle effettuate in TestConoscenzeValutazione, in quanto verrà aggiornato il layout in base al risultato complessivo ottenuto dall'utente; l'unica differenza si presenta nella modalità di inserimento dei nuovi risultati nel database. In questo caso viene preso dal database l'argomento corrente con il metodo getArgumentsByNameArgument(argomento corrente) e si confronta il valore salvato nella proprietà score, con il nuovo punteggio ottenuto dall'utente; nel caso in cui l'utente ha superato il test e il punteggio ottenuto è migliore di quello precedentemente salvato, verranno effettuati due inserimenti al database: uno per aggiornare il punteggio dell'argomento corrente ed uno per settare la proprietà unlocked dell'argomento successivo ad uno, mentre nella possibilità che l'utente abbia ottenuto un punteggio migliore rispetto al precedente, ma l'argomento successivo è già sbloccato, allora verrà effettuato un solo inserimento al database per aggiornare il nuovo score dell'argomento corrente. Una volta premuto il pulsante Home ,si viene portati di nuovo al fragment Home.

```
completamentoQuizViewModel.getArgumentByNameArgument(args1.codArgomento).observe(viewLifecycleOwner, Observer { it->

    Log.i(tag: "CompletamentQuizFrgment", msg: "L'argomento passato è: $it")
    //effettuo l'operazione se e solo se la funzione insertNewValue non è stata ancora eseguita
    if(completamentoQuizViewModel.controllochiamatealdatabase)
        completamentoQuizViewModel.insertNewValue(args1.numerorisposteesatte,it,args1.listargomenti.toList())

})
```

Figura 30-chiamata a database con argomento

Glossary

Questo fragment viene acceduto dal navigation drawer premendo la scelta glossario e svolge le stesse funzioni presenti nel fragment Home; in quanto tramite un listener unico, prende la descrizione del bottone premuto e naviga al fragment successivo, Teoria, passando l'argomento descritto nel bottone e una lista <Teoria>.

Teoria

Questo fragment illustra il contenuto della proprietà dell'oggetto Teoria, relativo all'argomento passato.

Achievement

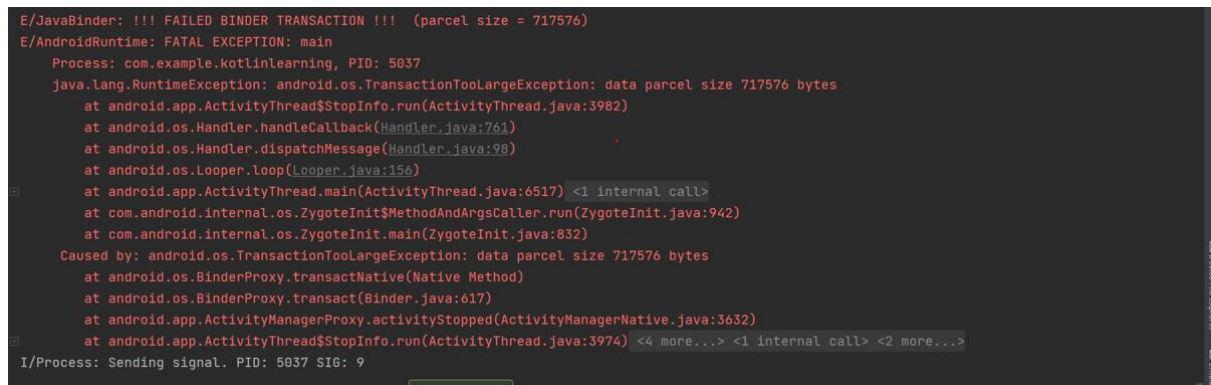
Questo fragment ha il compito di visualizzare i punteggi ottenuti dall'utente nei relativi argomenti, questo viene fatto osservando una lista di argomenti ottenuta dal database utilizzando la funzione

getAllArgument; quindi per ogni argomento verrà settato il nuovo punteggio ottenuto nella textview e verrà aggiornata la relativa progressbar.

2.5.3 Problematiche risolte

Nel corso dello sviluppo del nostro progetto abbiamo avuto diverse problematiche, che siamo riusciti a risolvere nel seguente modo:

1. La prima problematica che abbiamo affrontato riguardava la visualizzazione nel layout di dati ottenuti dal database, il contenuto veniva recuperato correttamente, ma non veniva visualizzato, questo perché l'operazione al database si concludeva dopo la creazione del layout. Per questo motivo abbiamo deciso di effettuare le chiamate a database, per oggetti che dovevano essere mostrati a display, nel fragment precedente, come spiegato sopra.
2. La seconda problematica è stata più difficile da individuare e risolvere. L'errore che presentavamo era il seguente, figura



```
E/JavaBinder: !!! FAILED BINDER TRANSACTION !!! (parcel size = 717576)
E/AndroidRuntime: FATAL EXCEPTION: main
Process: com.example.kotlinlearning, PID: 5037
java.lang.RuntimeException: android.os.TransactionTooLargeException: data parcel size 717576 bytes
    at android.app.ActivityThread$StopInfo.run(ActivityThread.java:3982)
    at android.os.Handler.handleCallback(Handler.java:761)
    at android.os.Handler.dispatchMessage(Handler.java:98)
    at android.os.Looper.loop(Looper.java:156)
    at android.app.ActivityThread.main(ActivityThread.java:6517) <1 internal call>
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:942)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:832)
Caused by: android.os.TransactionTooLargeException: data parcel size 717576 bytes
    at android.os.BinderProxy.transactNative(Native Method)
    at android.os.BinderProxy.transact(Binder.java:617)
    at android.app.ActivityManagerProxy.activityStopped(ActivityManagerNative.java:3632)
    at android.app.ActivityThread$StopInfo.run(ActivityThread.java:3974) <4 more...> <1 internal call> <2 more...>
I/Process: Sending signal. PID: 5037 SIG: 9
```

Figura 31-Errore Binder Transaction

Questo errore si presentava esattamente dopo lo sbloccaggio continuativo di tre argomenti, quindi dopo l'esecuzione di tre volte di questo (argomento,quizbottono,quiztastiera, completamentoquiz e home) ciclo, quando si usciva dall'applicazione e si tentava di rientrare ed si otteneva lo stesso risultato indipendentemente dalla schermata in cui si usciva. La causa(nonostante non siamo tuttora sicuri) era da attribuire al buffer del Binder Transaction, che possiede un buffer limitato e condiviso per tutti le transazioni in esecuzione per lo stesso processo e immagazzina gli argomenti Parcelable, passati da una view all'altra. Essendo le operazioni elencate prima, in ciclo ripetuto più volte, gli oggetti Parcelable, passati tramite safe args, si accumulavano fino ad eccedere lo spazio consentito e nel momento in cui l'app cercava di recuperare i dati,al momento della sua ricreazione, questi erano troppo grandi e quindi non salvati o salvati erroneamente(presumiamo). Abbiamo risolto questo problema aggiungendo questo pezzo di codice nell'action che va da CompletamentoQuizFragment a HomeFragment, all'interno del nav_graph.

```
app:popUpTo="@+id/homeFragment"
app:popUpToInclusive="true"
```

Queste operazioni liberano lo stack dei fragment, che passando da uno altro si accumulano; così facendo al termine di ogni ciclo viene liberato lo stack fino al fragment Home compreso, eliminando anche i dati passati tramite safe args che non si accumuleranno più.

2.6 TESTING

Per la fase di testing abbiamo implementato due file, uno per l'instrumented test e uno per l'unit test.

Per i test instrumented abbiamo implementato tre classi:

La prima si occupa di testare la main activity, interagendo con il bottone "Variabili" e completando il test.

```
@Test
fun testBottoneVar() {
    onView(withId(R.id.bottone_variabili)).perform(click())
    onView(withId(R.id.quiz_nome_argomento)).check(matches(isDisplayed()))
    onView(withId(R.id.b_quiz)).perform(click())
    for (i in 0..3) {
        onView(withId(R.id.b_risp1)).perform(click())
    }
    onView(withId(R.id.risposta_da_tastiera)).perform(
        typeText( stringToBeTyped: "var"),
        viewActionsCloseSoftKeyboard()
    )
    onView(withId(R.id.b_conferma)).perform(click())
    onView(withId(R.id.immagine_punteggio)).check(matches(isDisplayed()))
    onView(withId(R.id.b_home)).perform(click())
    onView(withId(R.id.bottone_variabili)).check(matches(isDisplayed()))
}
```

La seconda si occupa di testare il drawer interagendo con le opzioni del proprio menù.

```
@Test
fun testDrawer(){
    onView(withId(R.id.drawerLayout)).perform(DrawerActions.open())
    onView(withId(R.id.navView)).perform(NavigationViewActions.navigateTo(R.id.glossaryFragment))
    Espresso.pressBack()
    onView(withId(R.id.drawerLayout)).perform(DrawerActions.open())
    onView(withId(R.id.navView)).perform(NavigationViewActions.navigateTo(R.id.achievementFragment))
}
```

La terza classe invece si occupa di un semplice test, che testa i primi bottoni della MainActivity2.

```

@Test
fun testBottoneSalta(){
    onView(withId(R.id.b_Salta)).perform(click())
}
@Test
fun testBottonePro(){
    onView(withId(R.id.b_Procedi)).perform(click())
}

```

Per i test unit, abbiamo implementato due classi che testano una funzione di due viewmodel diversi.

Il primo test correctAnswerTest gestisce la funzione correctAnswer del viewmodel QuizTastieraViewModel, che si occupa di controllare se la risposta data nell'EditText del quiz, inserito da tastiera è giusta.

Nel test per primo viene inserita una risposta corretta fittizia nella variabile risposte del viewmodel, poi usa la funzione correctAnswer contenuta nel viewmodel, che verifica che la stringa data come input sia uguale alla stringa contenuta in risposte, per poi aumentare la variabile nriscorrette del viewmodel di uno.

Il test poi verifica che la funzione abbia avuto successo controllando che il valore expected che è uguale ad uno, sia uguale al valore contenuto in nriscorrette.

```

@Test
fun correctAnswerTest() {
    quizTastieraViewModel.risposte="risposta corretta"
    val input:String="risposta corretta"
    val expected=1
    var rispostareg= quizTastieraViewModel.correctAnswer(input)
    var output= quizTastieraViewModel.nriscorrette
    assertEquals(expected,output)
}

```

Il secondo test setQuestionTest gestisce il funzionamento della funzione setQuestion del viewmodel QuizBottoneViewModel, che si occupa di gestire il funzionamento del quiz a bottoni, inserendo le risposte nei corrispondenti bottoni.

In questo test viene generato un oggetto fittizio di tipo DomandeMultiple che viene associato alla variabile domandaAttuale del viewmodel, poi le stringhe corrispondenti alle risposte della variabile vengono aggiunte ad un'altra variabile di tipo MutableList<String> di nome risposte; il test, infine, si occupa di verificare che le varie risposte siano state inserite nel modo corretto dentro la variabile risposte.

```

@Test
fun setQuestionTest(){
    quizBottoneViewModel.domandaAttuale= DomandeMultiple( id_domanda: 1, testo_domanda: "2+2=?",
        cod_argomento: "Addizioni", risposta_1: "3", risposta_2: "1", risposta_3: "5", risposta_giusta: "4")
    quizBottoneViewModel.risposte.apply { this: MutableList<String>
        clear()
        add(quizBottoneViewModel.domandaAttuale.risposta_1)
        add(quizBottoneViewModel.domandaAttuale.risposta_2)
        add(quizBottoneViewModel.domandaAttuale.risposta_3)
        add(quizBottoneViewModel.domandaAttuale.risposta_giusta)
    }
    assertEquals(quizBottoneViewModel.domandaAttuale.risposta_1 , quizBottoneViewModel.risposte[0])
    assertEquals(quizBottoneViewModel.domandaAttuale.risposta_2, quizBottoneViewModel.risposte[1])
    assertEquals(quizBottoneViewModel.domandaAttuale.risposta_3 , quizBottoneViewModel.risposte[2])
    assertEquals(quizBottoneViewModel.domandaAttuale.risposta_giusta,quizBottoneViewModel.risposte[3])
}

```

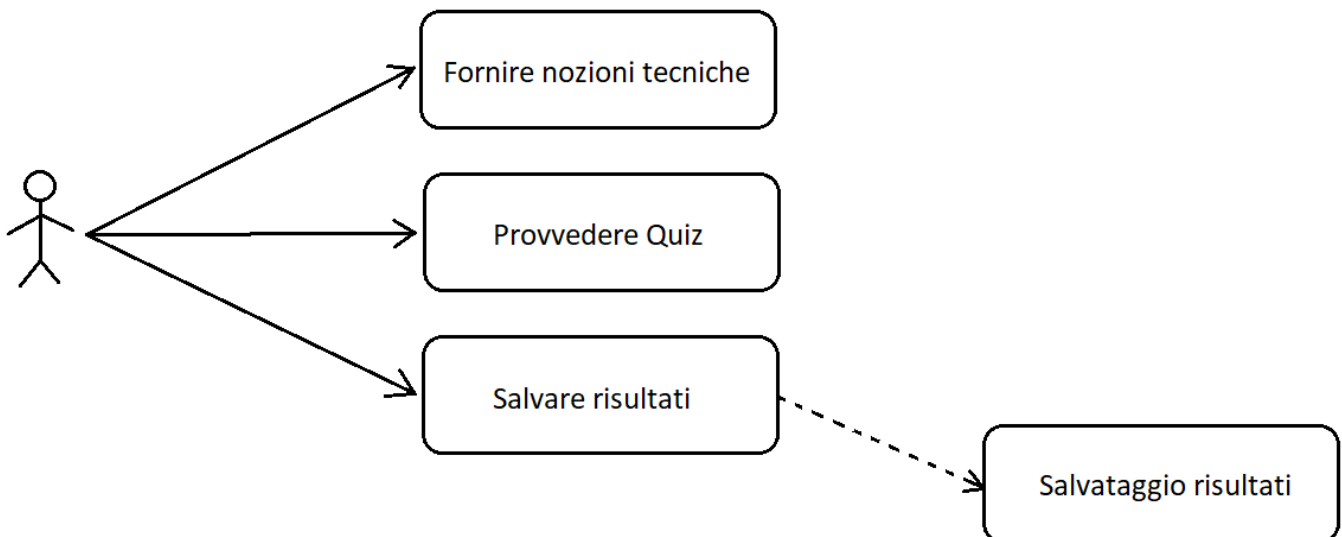
Note: durante il testing abbiamo riscontrato problemi poi risolti in entrambi i tipi di test, nel test Instrumented abbiamo risolto un problema downgradando la versione di espresso, mentre nell'unit test abbiamo incontrato un problema di mancata registrazione dell'istrumento che abbiamo risolto cambiando il runner dei test a Robolectric

3 SVILUPPO FLUTTER

3.1 REQUISITI E CASI D'USO

I requisiti per lo sviluppo dell'app in Flutter sono gli stessi dell'app sviluppata in Android. L'unica differenza visibile tra le due app si nota dal fatto che, l'app sviluppata in Flutter non prevede un test delle conoscenze, ma l'utente ha comunque subito accesso ai quiz riguardante il primo argomento e tutta la parte della teoria tramite il "Glossario".

Per il resto le app sono uguali sia nel loro funzionamento e che nel loro layout.



3.2 ARCHITETTURA DELL'APP FLUTTER

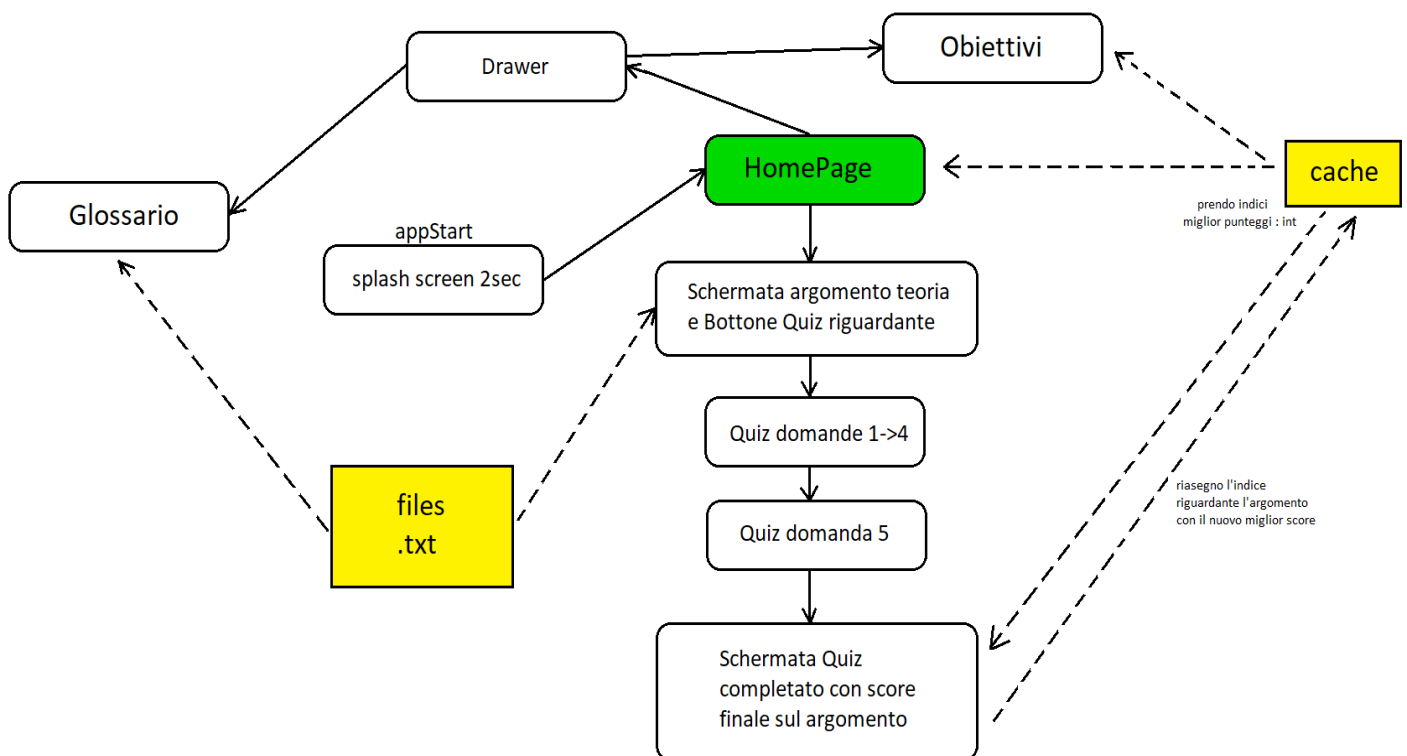
La Progettazione che abbiamo adottato funziona mediante la composizione di stateless widgets, stateful widgets e funzioni logiche

I dati non variabili, necessari per il funzionamento del display della teoria, sono inseriti in 9 file ".txt", uno per ogni argomento trattato nell'app.

I dati variabili, cioè i dati riguardanti i vari punteggi ottenuti nel completamento dei quiz, vengono salvati tramite il metodo Shared Preferences nella cache dell'app.

I quiz sono costruiti prendendo le informazioni da altri 9 file ".dart" contenenti le domande e le risposte sotto forma di `List<Map<String, Object>`.

Grafico illustrativo del funzionamento dell'app.



3.3 SVILUPPO DELL'APP

SplashScreen.dart:

All'avvio viene mostrata l'immagine KotlinLogo per due secondi per poi passare alla classe HomePage

HomePage.dart:

Listview con serie di widget children annidati, impostati al fine di ottenere lo stesso layout dell'app sviluppata in Android.

Vengono dichiarate le variabili bool "QuizNonCompletato" e i counter di tipo int, che vengono settati prendendo il valore salvato tramite il metodo Shared Preferences riportando 0 se null.

Si esegue la verifica dei counter per lo sblocco dei vari bottoni, se il counter legato ad un argomento e maggiore o uguale a tre allora l'argomento successivo sarà sbloccato cioè il bottone

```

void loadCounters() async {
  final prefs = await SharedPreferences.getInstance();
  setState(() {
    countervariabili = (prefs.getInt('scorevariabili') ?? 0);
    counterStringhe = (prefs.getInt('scoreStringhe') ?? 0);
    counterCondizioniECicli = (prefs.getInt('scoreCondizioniECicli') ?? 0);
    counterfunzioni = (prefs.getInt('scorefunzioni') ?? 0);
    counterNullSafety = (prefs.getInt('scoreNullSafety') ?? 0);
    counterArrayCollections = (prefs.getInt('scoreArrayCollections') ?? 0);
    counterClassi = (prefs.getInt('scoreClassi') ?? 0);
    counterEreditarieta = (prefs.getInt('scoreEreditarieta') ?? 0);
  });
}
  
```

dell'argomento successivo sarà reso premibile.

Quando viene premuto un bottone parte un NavigatorPush verso la classe ArgteoBtn dentro il file SchermataArgomentoTeoriaconBottoneQuiz.dart passandogli le due variabili richieste 'argomento' e 'teoria' sotto forma di stringa.

E' presente anche il drawer per avere accesso al Glossario e ai Obbiettivi.

```
Navigator.push(
  context,
  MaterialPageRoute(
    builder: (context) => ArgTeoBtn(
      argomento: "VARIABILI",
      teoria: 'variabili'))); // Arg
setState(() {
  if (countervariabili >= 3) {
    QuizVariabiliNonCompletato = false;
  }
}
```

SchermataArgomentoTeoriaconBottoneQuiz.dart:

Prende le due variabili passate come stringhe.

Mette la variabile 'argomento' e la mette nel text widget come titolo dell'argomento.

Dichiara una variabile 'data' vuota di tipo string. In base alla variabile 'teoria' si apre il file .txt

```
Container(
  alignment: Alignment.centerLeft,
  height: 80,
  padding: EdgeInsets.only(top: 50, left: 18 ),
  child: Text("${widget.argomento}", style: TextStyle(fontSize: 25),),
), // argomento // Container
```

collegato e si inizializza 'data' copiando le informazioni sotto forma di stringa. 'data' viene poi inserita nel text widget per fare il display della teoria.

Premendo il bottone Quiz si passa a Quiz.dart passandogli come variabili la variabile di tipo stringa ottenuta prima da Homepage

```
//prendo la teoria sotto forma di stringa dai file .txt in base al bottone premuto nella homepage
fetchFiledata() async {
  String dataFromtxt;
  dataFromtxt = await rootBundle.loadString('assets/TextFiles/'+'${widget.teoria}'+'.txt');
  setState(() {
    data = dataFromtxt;
  });
}
Navigator.push(
  context,
  MaterialPageRoute(
    builder: (context) => Quiz(argomentoquiz: widget.teoria));
```

Quiz.dart:

Importa i 9 file.dart dei modelli dei quiz contenenti 4 domande e una domanda5 con le relative risposte sotto forma di List<Map<String, Object>.

```
import 'modelliQuiz/QuizVariabili.dart' as variabili;
```

Crea una nuova variabile "List<Map<String, Object> a" e la riempi con le domande e le risposte prima importate in base alla variabile string 'argomentoquiz' ottenuta dalla schermata precedente.

Il layout della schermata quiz è un container widget contenente la domanda e 4 bottoni con le risposte.

```
List<Map<String, Object>> fetchdata() {  
  List<Map<String, Object>> a;  
  
  if (widget.argomentoquiz == 'variabili') {  
    a = variabili.domanda;  
  }  
}
```

Vengono mostrate la domanda e le risposte con un ciclo in base ad indexDomanda che va da 0 a 3, tenendo conto delle risposte giuste date.

```
child: Text(  
  argomento[indexDomanda]['domanda'], //prendo il testo della domanda dalla lista di map  
  textAlign: TextAlign.center,  
)), // Text, Center, Container  
- SizedBox(height: 30),  
...(argomento[indexDomanda]['risposte'] //itero le risposte dalla lista di map  
  as List<Map<String, Object>>)  
  .map(  
    (risposta) => Risposta(  
      risposta: risposta['risposte'],  
      answerTap: () {  
        if (rispostaInserita) {return;} //do la possibilita di scegliere una risposta una sola volta  
  
        rispostaData(risposta['score']); //rilevo il tap della risposta e prendo lo score  
        //se score=true risposta giusta  
        //se score=false risposta sbagliata  
  
        prossimaDomanda(); //passo alla prossima domanda  
      }  
    )  
  )
```

Le 4 risposte per ogni domanda sono un ciclo della classe Risposta in Domande1-4.dart.

Alla fine delle 4 domande con risposta da bottone si passa alla classe Q5

Contenuta in Domanda5.dart contenente la domanda5 passandogli argomentoquiz e scoreparziale.

Domanda5.dart :

Mostra a schermo la domanda5 in base ad "argomentoquiz" e prende la risposta da tastiera e verifica se la risposta è giusta o no e salva i dati se "scoreparziale" e' piu grande dello score salvato

```
//prende la key dello score salvato nella cache e riassiagna il valore  
void save(String score, int scoreparziale) async {  
  final prefs = await SharedPreferences.getInstance();  
  prefs.setInt(score, scoreparziale);  
}
```

```
void SaveData(String argomentoquiz, int scoreparziale) {  
  if (widget.argomentoquiz == 'variabili' && scoreparziale > countervariabili) {  
    save('scorevariabili', scoreparziale);  
  }  
}
```

Una volta inserita la risposta da tastiera e cliccato conferma si va all'ultima schermata "SchermataQuizFinito.dart" passandogli come variabile int 'scoretotale'

SchermataQuizFinito.dart:

```
setState(() {  
  rispostaDataastiera = _controller.text;  
});  
  
if(rispostaDataastiera == argomento[0]['risposta5'] || rispostaDataastiera == argomento[0]['risposta5bis']){  
  widget.scoreparziale++;  
}  
  
String arg = widget.argomentoquiz;  
int scoretotale = widget.scoreparziale;  
SaveData(arg, scoretotale);  
  
Navigator.push(  
  context,  
  MaterialPageRoute(  
    builder: (context) => SchermataQuizFinito(scoretotale: scoretotale))), // MaterialPageRoute
```

Prende la variabile "scoretotale" dalla classe Q5, mostra lo score a schermo con un'immagine in base allo score.



