23 MAGGIO 2018

PROJECTOR

PROGRAMMAZIONE PER DISPOSITIVI MOBILI (AA 2017-2018)

LAMBERTO BASTI
UNIVERSITÀ DI TORINO
Dipartimento di informatica

SOMMARIO

PREFAZIONE

Ringrazio il professor Damiani per l'opportunità :)

Projector è una applicazione per Android sviluppata per il corso di Programmazione per dispositivi mobili, materia del percorso di Laurea Magistrale in Informatica dell'Università di Torino.

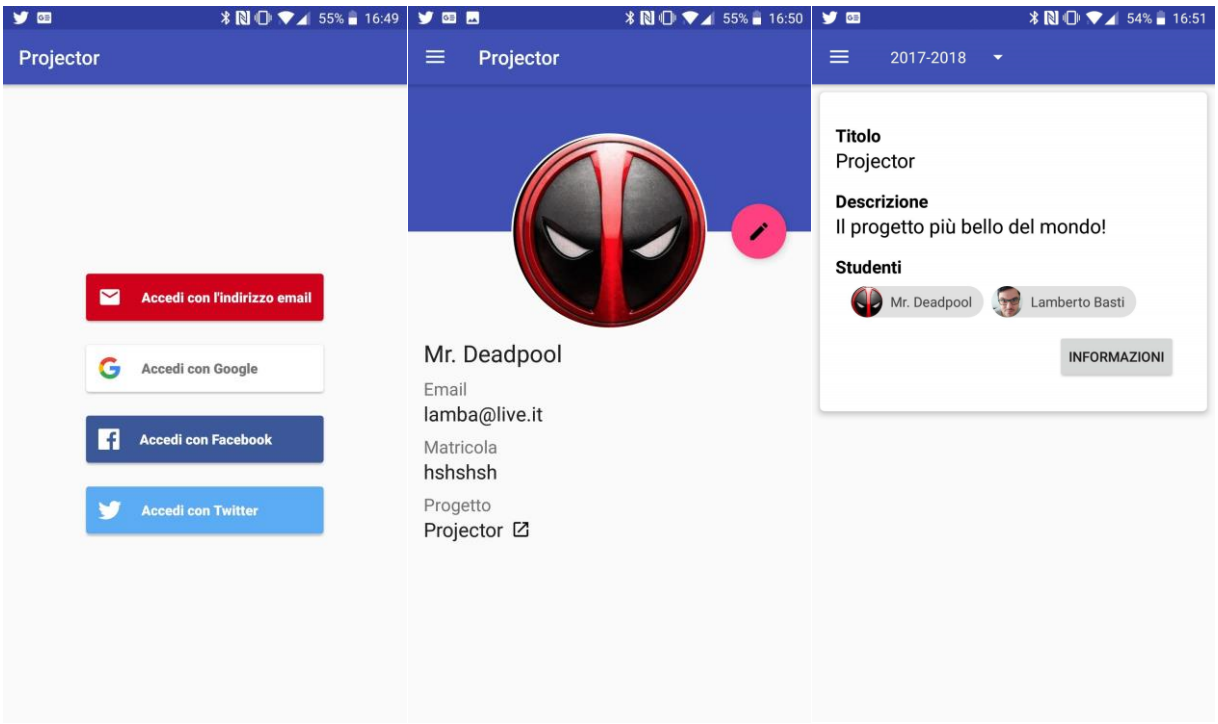
È stata sviluppata da Lamberto Basti ed è stata scritta in Kotlin, utilizzando Firebase come backend; entrambi verranno discussi più avanti.

Lo scopo di questa applicazione è mostrare la semplicità d'utilizzo di un approccio reattivo e funzionale tramite Kotlin e di come esso si sposa perfettamente con l'architettura di una applicazione Android, ma non solo. Sempre tramite approccio reattivo sono state costruite le API per utilizzare il Realtime Database di Firebase che permettono di gestire con facilità eventi asincroni come letture e scritture su database.

Ma a cosa serve Projector?

L'idea dietro questa app è semplificare la creazione di gruppo per i progetti di corsi universitari permettendo di registrarsi, crearsi un profilo personale, creare un profilo di un progetto e pubblicarlo invitando amici e cercandone di nuovi. Il tutto utilizzando tecnologie moderne e all'avanguardia.

Sfortunatamente, essendo da solo non sono riuscito ad implementare tutte le features che avevo progettato, come ad esempio notifiche push native e via e-mail, storaging di file aggiuntivi al progetto e tanto altro. Spero che possa comunque essere d'ispirazione per altri ragazzi ad utilizzare Kotlin e Firebase.



PERCHÉ KOTLIN?

Kotlin è un nuovo linguaggio di programmazione sviluppato da **JetBrains**, il produttore di alcuni dei migliori ambienti di sviluppo per diversi linguaggi. È un linguaggio pragmatico e conciso e rende l'esperienza di coding più efficiente e soddisfacente. Non ci sono ragioni per cui Java sia meglio di Kotlin ma ve ne sono molte per cui Kotlin è meglio Java, eccone alcune:

0) Interoperabilità con Java

Kotlin è 100% interoperabile con Java. È persino possibile recuperare un vecchio progetto in Java e continuare a svilupparlo in Kotlin. Tutti i principali Framework per Java sono quindi compatibili.

1) Sintassi familiare

Kotlin non è uno strano linguaggio nato in ambito accademico. La sua sintassi è familiare a qualsiasi sviluppatore proveniente da un dominio orientato agli oggetti e può essere compreso più o meno al volo. Vi sono ovviamente delle differenze rispetto a Java come ad esempio i costruttori o le dichiarazioni di variabili con `val` e `var`. Esempio:

```
class Foo {

    val b: String = "b"      // val means unmodifiable
    var i: Int = 0           // var means modifiable

    fun hello() {
        val str = "Hello"
        print("$str World")
    }

    fun sum(x: Int, y: Int): Int {
        return x + y
    }

    fun maxOf(a: Float, b: Float) = if (a > b) a else b

}
```

2) Interpolazione di stringhe

Permette una versione più smart e leggibile del Java `String.format()`:

```
val x = 4
val y = 7
print("sum of $x and $y is ${x + y}")   // sum of 4 and 7 is 11
```

3) Inferenza dei tipi

Kotlin inferirà i tipi ogni volta che ritieni migliori la leggibilità del codice:

```
val a = "abc"                           // type inferred to String
val b = 4                               // type inferred to Int

val c: Double = 0.7                     // type declared explicitly
val d: List<String> = ArrayList()       // type declared explicitly
```

4) **Smart cast**

Il compilatore Kotlin terrà traccia della logica del tuo programma ed eseguirà il cast dei tipi quando possibili, il che implica niente più controlli `instanceof` seguiti da cast espliciti:

```
if (obj is String) {
    print(obj.toUpperCase())      // obj is now known to be a String
}
```

5) **Uguaglianze intuitive**

Basta chiamare `equals()` esplicitamente, ora l'operatore `==` controlla anche la struttura dell'oggetto:

```
val john1 = Person("John")
val john2 = Person("John")
john1 == john2    // true  (structural equality)
john1 === john2   // false (referential equality)
```

6) **Argomenti predefiniti**

Non c'è più necessità di definire diversi metodi con argomenti differenti:

```
fun build(title: String, width: Int = 800, height: Int = 600) {
    Frame(title, width, height)
}
```

7) **Argomenti con nominativo**

Combinati con gli argomenti predefiniti, eliminano la necessità di diversi *builders*:

```
build("PacMan", 400, 300)                          // equivalent
build(title = "PacMan", width = 400, height = 300)  // equivalent
build(width = 400, height = 300, title = "PacMan")  // equivalent
```

8) **L'espressione when**

Lo switch è sostituito da un costrutto molto più flessibile e leggibile:

```
when (x) {
    1 -> print("x is 1")
    2 -> print("x is 2")
    3, 4 -> print("x is 3 or 4")
    in 5..10 -> print("x is 5, 6, 7, 8, 9, or 10")
    else -> print("x is out of range")
}
```

funziona sia come espressione che come dichiarazione, con e senza argomenti:

```
val res: Boolean = when {
    obj == null -> false
    obj is String -> true
    else -> throw IllegalStateException()
}
```

9) **Proprietà**

Getter e Setter possono essere personalizzati e aggiungi ai campi pubblici, il che implica niente più getter e setter superflui:

```
class Frame {
    var width: Int = 800
    var height: Int = 600

    val pixels: Int
        get() = width * height
}
```

10) Le classi data

Sono dei POJO completi di `toString()`, `equals()`, `hashCode()` e `copy()`, ma a differenza di Java non richiedono 100+ righe di codice:

```
data class Person(val name: String,
                  var email: String,
                  var age: Int)

val john = Person("John", "john@gmail.com", 112)
```

11) Overload degli operatori

È possibile eseguire l'overload su un set predefinito di operatori per migliorarne la leggibilità:

```
data class Vec(val x: Float, val y: Float) {
    operator fun plus(v: Vec) = Vec(x + v.x, y + v.y)
}

val v = Vec(2f, 3f) + Vec(4f, 1f)
```

12) Destrutturazione di oggetti

Alcuni oggetti possono essere destrutturati. Ad esempio, le mappe possono essere iterate così:

```
for ((key, value) in map) {
    print("Key: $key")
    print("Value: $value")
}
```

13) I range

Pura leggibilità:

```
for (i in 1..100) { ... }
for (i in 0 until 100) { ... }
for (i in 2..10 step 2) { ... }
for (i in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

14) Extension functions

Permette di aggiungere dei metodi a librerie statiche di sola lettura:

```
fun String.replaceSpaces(): String {
    return this.replace(' ', '_')
}

val formatted = str.replaceSpaces()
```

La libreria standard di Kotlin estende diversi oggetti largamente conosciuti in Java come ad esempio:

```
str.removeSuffix(".txt")
str.capitalize()
str.substringAfterLast("/")
str.replaceAfter(":", "classified")
```

15) Null safety

Kotlin fa distinzione fra variabili che ammettono `null` come valore e variabili che non ammettono `null` come valore:

```
var a: String = "abc"
a = null                // compile error

var b: String? = "xyz"
b = null                // no problem
```

Kotlin ti obbliga a gestire le NPEs ogni volta che si fa accesso ad una variabile che ammette `null`:

```
val x = b.length        // compile error: b might be null
```

Anche se ad un primo approccio sembra macchinoso, è una funzionalità estremamente comoda grazie anche allo smart cast da variabili nullabili a non nullabili:

```
if (b == null) return
val x = b.length        // no problem
```

C'è anche la possibilità di usare la chiamata sicura `?.`, che ritorna null se l'oggetto è nullo, altrimenti procede con l'esecuzione:

```
val x = b?.length       // type of x is nullable Int
```

Le safe calls possono essere messe in catena permettendo di evitare fastidiosi check `if else`, inoltre se si desidera ritornare qualcosa di diverso da `null` è possibile utilizzare l'operatore `?:`:

```
val name = ship?.captain?.name ?: "unknown"
```

Se nessuno di questi costrutti gestisce il tuo problema ed è assolutamente necessario lanciare un NPE è possibile farlo tramite l'operatore `!!`:

```
val x = b?.length ?: throw NullPointerException()  // same as below
val x = b!!.length                                 // same as above
```

16) Lambda migliorate

Kotlin offre un sistema di labda perfettamente bilanciato fra leggibilità e concisione grazie ad un design del linguaggio molto furbo. La sintassi è semplice:

```
val sum = { x: Int, y: Int -> x + y }    // type: (Int, Int) -> Int
val res = sum(4,7)                       // res == 11
```

Ecco i punti forti:

- Le parentesi tonde di un metodo possono essere omesse se la labda è l'unico o l'ultimo argomento.
- Se decidiamo di non dichiarare l'argomento di una lambda a singolo argomento, esso verrà implicitamente chiamato `it`:

```
numbers.filter({ x -> x.isPrime() })
numbers.filter { x -> x.isPrime() }
numbers.filter { it.isPrime() }
```

Tutto ciò rende possibile scrivere codice conciso e funzionale ed estremamente leggibile:

```
persons
    .filter { it.age >= 18 }
    .sortedBy { it.name }
    .map { it.email }
    .forEach { print(it) }
```

Il sistema di lambda di Kotlin unito alle extension functions rende il linguaggio ideale per la creazione di DSL. Un esempio è Anko, un DSL per Android sviluppato da JetBrains.

17) Supporto IDE

Vi sono diversi IDE che supportano Kotlin, ma il più raccomandato è IntelliJ IDEA (e di conseguenza Android Studio) che supportano Kotlin out of the box. Un esempio del perché scegliere IntelliJ come IDE per scrivere Kotlin è la conversione automatica di codice Java in Kotlin quando viene incollato.

Queste sono solo alcune delle features che Kotlin offre e nel mio progetto ho utilizzato buona parte di esse ed altre ancora. Per chi proviene da Java, nulla vieta di scrivere codice Kotlin in pieno stile Java, ma i tool messi a disposizione dal linguaggio per la risoluzione di problemi ricorrenti, come NPEs e callbacks, sono intrinsecamente più potenti di quelli offerti da Java, portando lo stile del codice a migrare con naturalezza verso una programmazione più reattiva e funzionale.

FIREBASE

UN PO' DI STORIA

Ad ottobre 2014 Google acquisì una azienda chiamata Firebase. Ai tempi, Firebase aveva creato un database NoSQL ramificato, molto simile ad un documento JSON, dove è possibile mettersi in ascolto su diverse rami o foglie in attesa di modifiche, cancellazioni o creazione di nuovi dati. Tali hook triggerano sui client in ascolto del codice a cui viene fornita una istantanea dei dati modificati sul database.

Una architettura del genere si presta perfettamente ad applicativi con UI dinamiche, in particolare nei dispositivi mobili dove collaborazione e socialità sono la chiave di volta del successo.

OGGI

Dal 2014 ad oggi molte cose sono state aggiunte nell'adesso MBaaS (Mobile Backend as a Service) Firebase. Innanzitutto, una volta acquisita da Google, l'intero backend di Firebase è stato spostato sui Google Cloud Services, poi sono stati aggiunti diversi prodotti targati fiamma gialla, alcuni recentissimi come Machine Learning Kit (ML Kit) aggiungo appena dopo il Google I/O 2018 tenuto a maggio. Da notare come i diversi prodotti possono essere integrati singolarmente nei propri progetti.

Firebase, ad oggi, offre 3 piani tariffari:

- **Spark**: gratuito, offre un generoso quantitativo di risorse senza versare un centesimo.
- **Flame**: 25$/mese, il classico piano flat, diverse funzionalità in più e soglie molto alte.
- **Blaze**: pagamento ad utilizzo.

Il piano Spark è più che sufficiente per lo sviluppo di una app da parte di un piccolo team, e se si fa un uso centellinato degli hook, anche per la prima parte di produzione.

Le APIs client sono disponibili per iOS, Android e JavaScript, le APIs admin per NodeJS, Java, Python e Go.

NB: solo su NodeJS è disponibile l'intero set di APIs admin.

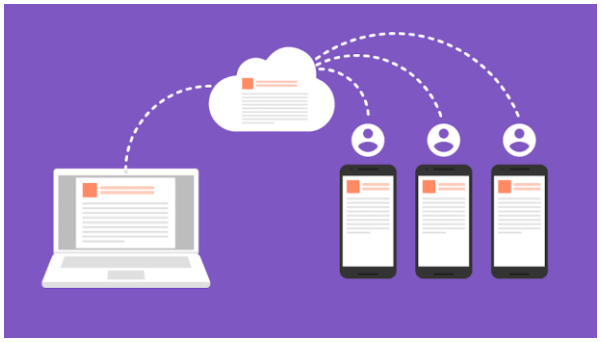
Per ragioni di brevità mi soffermo ad illustrare solo i prodotti che ho utilizzato. Per tutte le informazioni è possibile consultare l'intera suite a https://firebase.google.com.

AUTHENTICATION

Firebase Authentication offre APIs semplici e concise per l'autenticazione degli utenti. Permette l'interfacciamento out of the box con i sistemi OAuth2 di Google, Facebook e Twitter, oltre che al classico sistema e-mail/password. Molto interessanti sono il login anonimo e con il cellulare che non ho abilitato in Projector per questioni di tempo.

È possibile aggregare i provider di login, in modo da permettere all'utente di eseguire il login tramite i diversi social senza problemi.

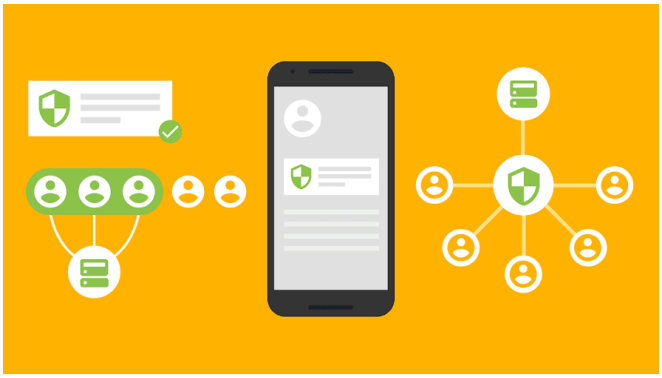
REALTIME DATABASE



Firebase RTD è un database NoSQL strutturato come un JSON in cui è possibile mettersi in ascolto di uno o più rami. Quando il database viene modificato tutti i listeners convolti nelle zone mutate vengono notificati con i dati aggiornati. Questo permette di aggiornare UI su diversi client alla modifica di dati sul RTD con semplicità estrema.

L'SDK di RTB permette di gestire in maniera automatizzata il caching dei dati in locale, ciò permette di risparmiare diversi MB preziosi nelle connessioni mobili.
Diretta conseguenza del caching è la possibilità di accedere ai dati locali in offline, permettendo l'utilizzo di funzionalità parziali dell'applicazione anche quando la connessione non è disponibile. Vi sono però delle accortezze da gestire in questo scenario, nel particolare per mantenere consistenti transazioni che necessitano di atomicità è necessario tenere traccia manualmente di tali transazioni eseguite offline per poi ritentarle quando online.

RTD offre inoltre un sistema di regole di sicurezza che permette di proteggere il database o parte di esso da letture e scritture da utenti non autorizzati. Le regole di sicurezza vengono espresse tramite un JSON e permette di esprimere regole elastiche e semplici ma estremamente potenti. L'utilizzo fatto di tali regole in Projector è stato minimale, sia perché non vi era reale necessità, sia perché per poterle strutturare in maniera scalabile è necessario progettare l'intera struttura dati a priori dello sviluppo dell'applicativo e per ragioni di tempo non ho approfondito l'argomento.



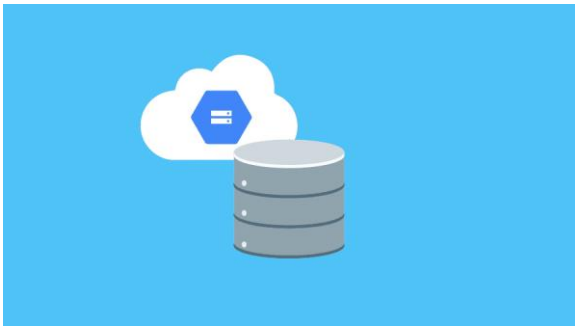
Il contro principale di RTB è la mancanza di un sistema di query strutturato. Questa mancanza porta alla necessità di mantenere il proprio albero dei dati il più piatto possibile, costringendo a costruire soluzioni di indexing e ridondare dati. Per venire in contro a tali problematiche, il team di Firebase ha creato un nuovo database chiamato **Firestore** che permette di strutturare i dati in cartelle che contengono documenti JSON. Ancora una volta per motivi di tempo non mi sono soffermato su questa nuova tecnologia ed ho scelto il "vecchio" RTD poiché avevo una già discreta confidenza con esso. Nel caso in cui ci si approcci alle tecnologie Firebase per la prima volta e vi sia necessita di un database, consiglio caldamente di ignorare RTD e utilizzare Firestore.

CLOUD FUNCTIONS

Firebase Cloud Functions è un sistema di microservizi. Una Firebase Cloud Function è una funziona scritta in JavaScript o TypeScript che viene eseguita ad un particolare trigger. Un trigger per una FCF può essere un evento più o meno complicato come ad esempio POST o GET HTTP, una scrittura su RTD, ecc.

In Projector ho fatto largo utilizzo delle Cloud Functions per semplificare lo smistamento dei dati scritti dai client. Ad esempio, quando un utente crea un progetto, tale progetto è privo di un ID, dell'anno in cui viene creato e del suo creatore; tale progetto viene scritto nel nodo radice `add_projects/` dove una function è in ascolto su un qualsiasi nodo creato all'indirizzo `add_projects/{uid}`. Tale function intercetta il dato, vi aggiunge l'anno di creazione, l'utente che lo ha creato come proprietario, genera poi un ID univoco e lo sposta nel nodo `projects/`.

Questa è solo una delle diverse functions che ho scritto, ma rende l'idea di come il database stesso è in grado di manipolare e ordinare i dati secondo una logica desiderata.

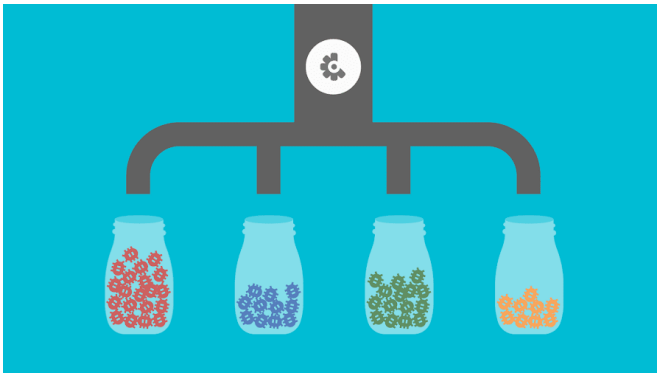


STORAGE

Figlio dei bucket dei Google Cloud Services, Firebase Storage ha come punto di forza la semplicità d'utilizzo delle sue APIs lato client che permettono di gestire i singhiozzi di una rete mobile e adattarsi in base alla velocità della connessione senza dover scrivere una singola riga di codice. Ha un sistema di regole di sicurezza simile a quello del RTD e si integra perfettamente con Firebase Authentication per la gestione dei diritti di accesso ai file.

CRASHLYTICS

È un servizio di crash reporting automatizzato. Una volta importate le dipendenze nel progetto, Crashlytics intercetta e trasferisce lo stack dell'eccezione incriminata, insieme a parte del logcat in Andorid, sulla console di Firebase accessibile tramite Web. Da quei è possibile tenere traccia dei principali problemi che portano a crashare l'applicazione ed intervenire prontamente non appena qualcosa va storto persino in produzione.



FIREBASE UI

Firebase UI è una libreria per iOS, Android e Web che permette di semplificare l'integrazione dei servizi Firebase con la propria applicazione. Contiene diversi template per la UI che riducono drasticamente i tempi di prototipazione.

PROJECTOR

Questa sezione è dedicata a Projector e alle sue principali funzionalità. Mi sono soffermato ad illustrare il codice più didatticamente significativo ma vi sono diverse classi interessanti, come ad esempio l'`ImagesAdapter` che ho scritto appositamente per questo progetto e poi spostato in una libreria a parte.

Molte componenti utilizzate fanno parte dell'SDK Android come `RecyclerView`, `Spinner`, `FloatingActionButton`. In caso non conosceste alcune di queste librerie, consiglio di dare una sguardo alle loro sezioni su sito https://developer.android.com/.

STRUTTURA DATI

LATO CLIENT

Per modellare i dati necessari all'app mi sono servito di due semplici classi `data`, `User` e `Project`.

```
import com.google.firebase.database.Exclude

data class User(@get:Exclude @set:Exclude var uid: String = "",
                var name: String? = null, var surname: String? = null,
                @get:Exclude @set:Exclude var isAdmin: Boolean = false,
                var imgUri: String? = null, var email: String? = null,
                var projectId: String? = null, var badgeNumber: String? = null){

    @Exclude
    fun getDisplayName(): String {
        return "$name $surname"
    }
}

data class Project(@get:Exclude @set:Exclude var id: String? = null,
                   var title: String? = null,
                   var description: String? = null,
                   var owners: HashMap<String, Boolean>? = null,
                   var images: HashMap<String, Boolean>? = null,
                   var year: String? = null,
                   var repos: HashMap<String, Boolean>? = null,
                   var documents: HashMap<String, Boolean>? = null)
```

Qui entra in gioco la magia dell'overload di Kotlin. Queste poche righe di codice permettono di esprimere tutte le possibili combinazioni di costruttori che hanno tutti o solo parte di questi attributi, nullabili o meno. Se la sintassi non è chiara consiglio di fare un salto alla sezione Kotlin ad inizio relazione, per la precisione al punto #6.

Soffermatevi sula annotazione `@Exclude` importata dall'SDK di RTD. `@Exclude` permette, appunto, di escludere un attributo pubblico o dei getter e setter di un attributo privato dall'essere utilizzati da Firebase quando una istanza di questa classe viene data in pasto al RTD. Ciò significa che quegli attributi verranno ignorati sia quando devo "salvare" l'istanza sul RTD sia quando il RTD crea una istanza sul client locale. Di default tutti gli attributi di una `data` class sono privati per cui gli `@Exclude` vanno assegnati a getter e setter tramite una annotation built-in di Kotlin.

`getDisplayName()` di `User` è un metodo di comodità.

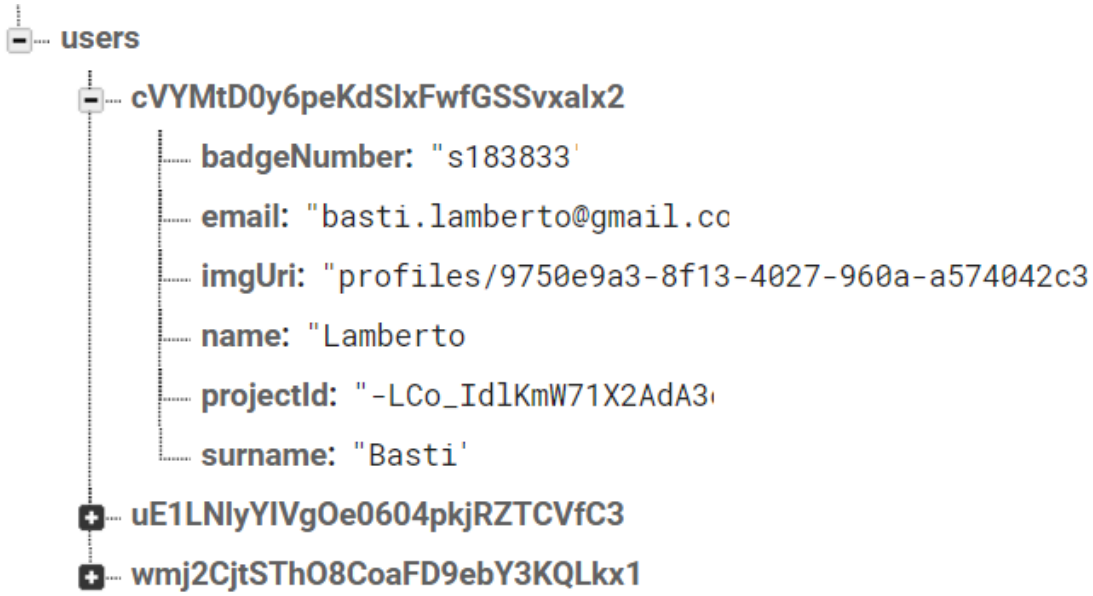
LATO DATABASE

Come accennato nella sezione precedente dedicata a Firebase, i dati nel RTD sono strutturati ad albero esattamente come in un JSON. Qui di seguito mostrerò uno istantanea dei dati permanenti nel backend a cominciare dagli indici:

Nel nodo `indexes/` vengono conservati una serie di metadati utili al corretto funzionamento di Projector. In ordine abbiamo le coppie chiavi-valore `<userId, true>` in `indexes/admins/` che permettono di identificare chi ha i diritti di amministratore; a seguire, nel nodo `email-uid` è presente un indice di look-up per recuperare l'identificato utente a partire dalla sua e-mail (i campi chiave del RTD non ammettono diversi caratteri tra cui "**.**" e "@" necessitando una codifica); infine vi è una lista degli anni scolastici in cui sono stati pubblicati dei progetti – a loro volta contenenti coppie chiave-valore `<projectId, true>` - e dei contatori di comodità.

Nei nodi `previousUserData/` e `previousProjects/` sono invece conservati la cronologia dei cambiamenti degli utenti e i progetti eliminati.

I progetti sono conservati nel RTD con una struttura simile alla rispettiva classe lato client. Notare come il campo `id` manca dal corpo del progetto grazie all'esclusione del rispettivo setter escluso dall'annotazione menzionata prima. A runtime infatti, l'`id` del progetto verrà recuperato dalla chiave con cui è stata eseguita la query ed aggiunta al progetto manualmente.



Similmente ai progetti, anche gli utenti sono conservati con la medesima tecnica. Infatti, i campi contrassegnati da `@Exclude` non sono presenti nel corpo dell'utente. I campi `id` e `isAdmin` vengono recuperati rispettivamente tramite chiave della query ed una query all'indice prima analizzato.

NB: una struttura dati sugli utenti come questa necessita di una rivisitazione a causa del nuovo GDPR imposto dall'Unione Europea.

CLOUD FUNCTIONS

Molti dati prima di essere disponibili agli utenti tramite il RTB necessitano di essere elaborati e controllati. Per evitare di includere questa logica in un fat client, ho spostato tali funzionalità sulle cloud functions (CF).
Le FCF sono scritte in TypeScript, una versione tipizzata (o quasi) di JavaScript a cura di Microsoft. Vediamo alcuni esempi per comprenderne meglio il funzionamento.

FUNCTION PROCESSNEWPROJECT

```
export const processNewProject = functions.database
    .ref('add_projects/{uid}')
    .onCreate((snapshot: DataSnapshot, context: EventContext) => {
        const currentDate = new Date();
        const currentYear = currentDate.getFullYear();
        const userId = context.params.uid;
        let projectYear: string;
        let otherYear: number;
        if (currentDate.getMonth() >= 9) {
            otherYear = currentYear + 1;
            projectYear = currentYear + "-" + otherYear
        } else {
            otherYear = currentYear - 1;
            projectYear = otherYear + "-" + currentYear
        }
        const project = snapshot.val();
        project.year = projectYear;
        const ref = admin.database().ref('projects').push(project);
        ref.then((value: any) =>{
            return admin.database().ref('add_projects/' + userId)
                .set(null).then((useless) => {
                return admin.database().ref('indexes/total-projects')
                        .once('value',totalProjects => {
                        if (totalProjects.val() !== undefined) {
                            return admin.database()
                                .ref('indexes/total-projects')
                                .set(totalProjects.val() + 1);
                        } else return admin.database()
                            .ref('indexes/total-projects').set(1);
                });
            });
        });
        return ref
    });
```

`processNewProjcet` è in ascolto su ogni nuovo nodo creato sul ramo `add_projects/{uid}`. È proprio in quel ramo che i nuovi progetti vengono scritti prima di essere intercettati, modificati e posizionati correttamente. Una FCF infatti consiste in una lambda di callback i cui parametri variano in base all'evento che la triggera. In questo caso abbiamo come parametro un `DataSnapshot` contente chiave e dato scritto e un `EventContext` contenente dati di contesto utili.

Il flusso di esecuzione della callback consiste in una serie di `promise` concatenate. Ciò che qui viene fatto è completare i dati mancanti del progetto aggiungendo l'anno scolastico corrente e l'`id` dell'utente che lo ha creato. Una volta pronto, il progetto completo viene scritto nel nodo `projects/` con una chiave univoca generata dal metodo `.push(project)`. Al completamento della `promise` `ref`, viene cancellato il progetto da `add_projects/` e aggiornato il contatore dei progetti totali `indexes/total-projects` tenendo conto di un suo possibile `undefined`.

FUNCTION ADDPROJECTTOPROFILE

```
export const addProjectToProfile = functions.database
    .ref('projects/{pid}/owners/{uid}')
    .onCreate((snapshot, context) => {
        return admin.database().ref('users/'
            + context.params.uid + "/projectId")
            .set(context.params.pid);
    });
```

È in ascolto dell'aggiunta di nuovi partecipanti ad un progetto. La funzione è molto semplice, intercetta gli identificativi del progetto e dell'utente aggiunto grazie alla sintassi parametrica del riferimento nel RTD e scrive nel nodo `users/{uid}/projectId/` il valore `{pid}`.

FUNCTION CREATEINDEXUSER E UPDATEINDEXUSER

```
export const createIndexUser = functions.database
    .ref('users/{uid}/email')
    .onCreate((snapshot: DataSnapshot, context: EventContext) => {
        const email: string = snapshot.val();
        return admin.database().ref('indexes/email-uid/'
            + encodeEmail(email)).set(context.params.uid)
    });
export const updateIndexUser = functions.database
    .ref('users/{uid}/email')
    .onUpdate((snapshot: Change<DataSnapshot>, context: EventContext) => {
        const newEmail: string = snapshot.after.val();
        const oldEmail: string = snapshot.before.val();
        return admin.database()
            .ref('indexes/email-uid/' + encodeEmail(newEmail))
            .set(context.params.uid,
                (error: Error) => {
                    admin.database().ref('indexes/email-uid/'
                        + encodeEmail(oldEmail)).set(null)
                }).then((value) => {
                    return admin.database().ref('previousUserData/'
                        + context.params.uid + "emails").push(oldEmail)
            })
    });
function encodeEmail(s: string): string {
    return s
        .replace(new RegExp('\\.', 'g'), '%2E')
        .replace(new RegExp('@', 'g'), '%40');
}
```

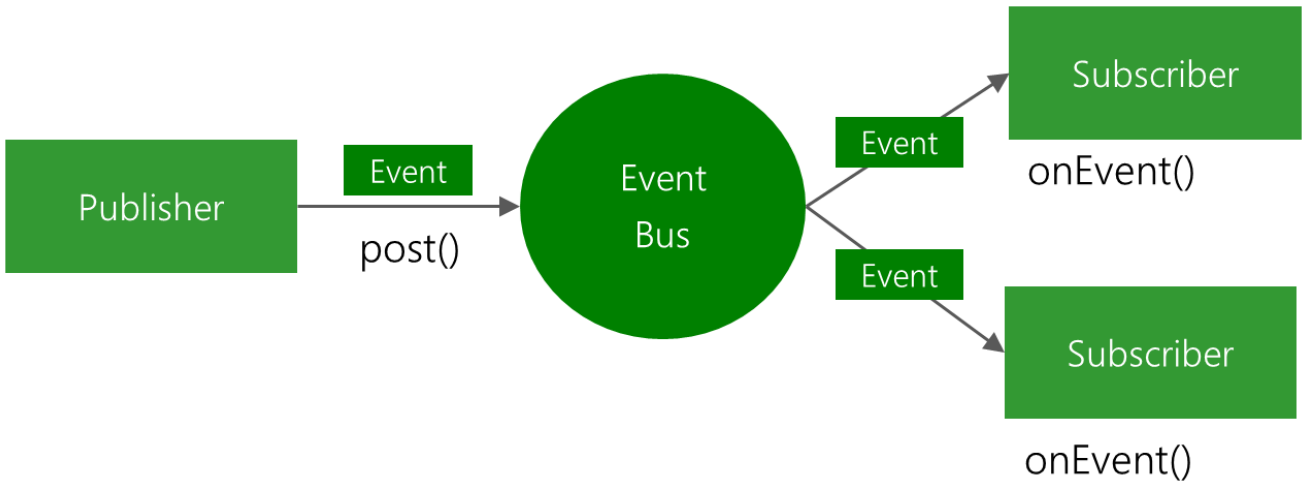
Queste due FCFs si occupano di creare l'indice di look-up da e-mail a identificativo utente.

`createIndexUser` è in ascolto della creazione del campo `email` in un utente qualsiasi. Quando viene aggiunto il campo, l'email viene codificata e viene scritta la coppia `<e-mail, userId>` nel nodo `indexes/email-uid/`.

Similmente, la seconda funzione è in ascolto dei cambiamenti del campo `users/{uid}/email`; ad ogni cambiamento, oltre ad aggiornare l'indice corrispondente, la precedente e-mail viene salvata nel nello storico delle email dell'utente tramite un `.push(oldEmail)` in `previousUserData/{uid}/emails/`.

GREENROBOT EVENTBUS

EventBus è una libreria open-source per Android e Java che usando il pattern publisher/subscriber permettere l'accoppiamento sciolto fra componenti. EventBus permette di centralizzare la comunicazione fra classi disaccoppiate con appena qualche riga di codice e velocizza molto lo sviluppo di una app.



QUICKSTART GUIDE

Step 1) Definire un evento

Gli eventi sono dei semplici POJO senza particolari specifiche:

```
class MessageEvent(val message: String)
```

Step 2) Preparare il destinatario (subscriber)

Un subscriber implementa dei metodi per gestire gli eventi (chiamati "subscriber methods") che vengono chiamati quando il metodo viene pubblicato. Questi metodi sono definiti con l'annotazione `@Subscribe`:

```
// This method will be called when a MessageEvent is posted (in the UI thread
for Toast)
@Subscribe(threadMode = ThreadMode.MAIN)
fun onMessageEvent(event: MessageEvent) {
    Toast.makeText(activity, event.message, Toast.LENGTH_SHORT).show()
}

// This method will be called when a SomeOtherEvent is posted
@Subscribe
fun handleSomethingElse(event: SomeOtherEvent) {
    doSomethingWith(event)
}
```

I subscriber devono inoltre registrarsi e annullare la registrazione dal bus quando necessario. Solo i subscriber registrati riceveranno gli eventi. In Android, activities e frammenti vanno registrati secondo il loro ciclo vitale. La maggior parte dei casi vengono registrati in `onResume` / `onPause`:

```
override fun onResume() {
    super.onResume()
    EventBus.getDefault().register(this)
}
override fun onPause() {
    EventBus.getDefault().unregister(this)
    super.onPause()
}
```

Step 3) Pubblica gli eventi

Pubblica un evento in una qualsiasi parte del codice. Tutti i subscribers correntemente registrati il cui evento corrisponde a quello pubblicato lo riceveranno:

```
EventBus.getDefault().post(MessageEvent("Hello everyone!"))
```

TRA I PRO DI EVENTBUS:

- semplifica la comunicazione fra componenti
- disaccoppia mittente e destinatario
- ottime prestazioni nella gestione della UI (ad esempio Activities e Fragments) e threads in background
- riduce la complessità, dipendenze inclini ad errori e problemi di life-cycle
- estremamente veloce ed efficiente
- pesa appena 50kbyte
- utilizzata da app con più di 100 milioni di downloads
- funzionalità avanzate come la selezione del thread, priorità dei destinatari, ecc.

OBJECT DATABASE

Projector comunica costantemente con il suo backend, letteralmente in qualsiasi momento tramite i realtime hooks offerti dalle APIs di RTD. Per semplificare l'accesso al RTD ho creato un particolare costrutto di Kotlin, un `object`, che contiene tutte le funzionalità per interfacciarsi ad esso permettendo di leggere e scrivere progetti, utenti, ecc. Qui di seguitò espongo e commento alcuni metodi che ritengo ottimi esempi didattici i quali permettono di comprendere meglio le meccaniche di una programmazione reattiva e funzionale.

COS'È UN OBJECT ?

object Database {...}

Un `object` in Kotlin è un oggetto che viene instanziato dalla JVM all'avvio del programma o al suo primo utilizzo (qualcuno ha detto Spring IoC?). Questo oggetto può ereditare una classe ed in quel caso vanno specificati dei parametri statici da passare come argomenti del costruttore genitore, altrimenti eredita automaticamente da `Any` (l'oggetto radice in Kotlin non è `Object`), come nel caso di `Database`, il cui costruttore non ha parametri. Una volta dichiarato, in qualsiasi parte del codice sarà possibile importarlo e ed utilizzarlo. Ricorda i Singleton in Java, solo che servono 5 caratteri invece di 20 righe di codice.

L'INTESTAZIONE DI DATABASE.KT

Ma torniamo a Projector. I metodi all'interno di `Database` non sono pochi ma sono simili. Ne mostro alcuni in modo tale da far comprendere le meccaniche d'utilizzo.

```
object Database {

    const val USERS = "users"
    const val TAG = "Database"
    const val PROFILE_PICTURES_PATH = "profiles"
    const val ADD_PROJECT_REF = "add_projects"
    const val PROJECT_IMAGES_PATH = "project/images"
    const val YEARS_INDEXED = "indexes/projects-by-years"
    const val PROJECTS = "projects"

    private val db = FirebaseDatabase.getInstance()
    private val auth = FirebaseAuth.getInstance()
    private val hookMap = HashMap<DatabaseReference, ValueEventListener>()
    private val storage = FirebaseStorage.getInstance()
…

}
```

Ecco l'intestazione dell'oggetto. Vi sono delle costante di comodità che utilizzo all'interno dell'oggetto e dei membri privati che contengono informazioni su Firebase. Da notare la variabile `hookMap` che contiene i `ValueEventListener`s eseguiti quando i dati sul RTD referenziati dai `DatabaseReference`s cambiano. La mappa serve per "scollegare" gli hook dal RTD quando viene eseguito il logout, ma lo vedremo meglio dopo.

METODO GETUSERBYUID()

```
fun getUserByUid(uid: String, onResult: (user: User?) -> Unit,
                         onFailure: (e: Exception) -> Unit){
    db.reference.child(USERS)
                .child(uid)
                .addListenerForSingleValueEvent(object: ValueEventListener{
        override fun onCancelled(p0: DatabaseError?) {
            p0?.toException()?.let { onFailure(it) }
        }

        override fun onDataChange(p0: DataSnapshot?) {
            if(p0?.getValue(User::class.java)?.apply {
                this.apply{
                    this.uid = p0.key
                    isUserAdmin(this.uid, {
                        this.isAdmin = it
                        onResult(this)
                    })
                }
            } == null) onResult(null)
        }
    })
}
```

Partiamo dagli argomenti:

- `uid: String`:
  Il codice identificativo univoco di un utente. Viene generato da Firebase Authentication ed è stato previamente recuperato da chi chiama questo metodo.
- `onResult: (user: User) -> Unit`:
  In Kotlin le funzioni posso essere variabili. Qui dichiaro un parametro come una funzione che riceve in ingresso uno `User` e ritorna `Unit` (il `void` di Kotlin).
- `onFailure: (e: Exception) -> Unit`:
  Un'altra lambda, Viene utilizzata quando qualcosa va storto per fornire l' `Exception` a chi ha chiamato `getUserById()`.

Nel corpo del metodo recupero la referenza al nodo che mi interessa, in questo caso `/users/uid` e fornisco alla referenza una implementazione anonima dell'interfaccia `ValueEventListener` dove in caso di fallimento a per errori di rete o mancati diritti di lettura, con le dovute accortezze sui possibili `null`, propago l'errore tramite la lambda `onFailure`, altrimenti controllo che l'utente ritornato esista e se esiste chiamo la callback `onResult` che eseguirà il codice del richiamante.

Da notare che Firebase gestisce letture e scritture tramite delle `AsyncTask`s.

METODO GETCURRENTUSER()

```
fun getCurrentUser(onResult: (user: User) -> Unit, hook: Boolean = false){
    getCurrentAuthUserID().apply {
        val userRef = db.reference.child("$USERS/$this")
        val valueEventListener = object : ValueEventListener{
            override fun onCancelled(p0: DatabaseError?) {}

            override fun onDataChange(p0: DataSnapshot?) {
                p0?.apply {
                    getValue(User::class.java)?.apply {
                        uid = p0.key
                        isUserAdmin(this.uid,{
                            this.isAdmin = it
                            onResult(this)
                        })
                    }?:onResult(User())
                }
            }
        }
        if(hook){
            hookMap[userRef] = userRef.addValueEventListener(valueEventListener)
        } else userRef.addListenerForSingleValueEvent(valueEventListener)
    }
}
```

Simile al metodo precedente ma permette di rimanere in ascolto sui cambiamenti che avvengono sull'utente correntemente autenticato. Da notare come questa volta l'oggetto `valueEventListener` instanziato viene poi aggiunto alla mappa degli hook solo se il parametro `hook` è settato a `true`.

Ogni volta che l'utente corrente cambierà alcuni dati personali, come ad esempio immagine profilo, nome, numero matricola, ecc, `valueEventListener.onDataChange()` verrà nuovamente chiamato dalle routines di Firebase e conseguentemente anche `onResult` verrà eseguito con il codice originariamente inserito dal chiamante di `getUserById()`. È in questo modo che aggiorno l'intestazione della barra di navigazione di Projector al cambio delle informazioni personali.

METODO CHECKCURRENTUSERINTEGRITY()

```
fun checkCurrentUserIntegrity(onResult: (isComplete: Boolean) -> Unit) {
    getCurrentUser ({ user ->
        if(user.name == null || user.surname == null ||
           user.name == ""   || user.surname == "") onResult(false)
        else if(!user.isAdmin && user.badgeNumber == null ||
                user.badgeNumber == "") onResult(false)
        else onResult(true)
    })
}
```

Riceve in ingresso una lambda che a sua volta ha in ingresso un booleano. Il metodo si occupa di controllare se un utente ha il profilo completo o meno, se si viene fornito `true` alla lambda, altrimenti `false`. Questo metodo chiamato all'avvio dell'app se si è autenticati ed è utilizzato per far inserire i propri dati appena dopo la registrazione, ovvero quando l'utente esiste come istanza in FirebaseAuth, ma non nel RTD, inoltre permette di coprire tutta una serie di possibili edge cases in cui è meglio non trovarsi se non si vuole annegare nelle linee di logcat in verbose.

METODO LOGOUT()

```
fun logOut(context: Context, onResult: () -> Unit) {
    for((key, value) in hookMap){
        key.removeEventListener(value)
    }
    AuthUI.getInstance().signOut(context)
            .addOnCompleteListener { onResult()}
}
```

Questo metodo permette di eseguire il logout e di essere notificati tramite una callback quando viene completato. Per evitare di incorrere in crash indesiderati o peggio ancora in memory leaks, è necessario chiudere tutti gli hook in ascolto. Se non venisse fatto, in caso di trigger Firebase eseguirà del codice su oggetti ormai deallocati appartenenti alla precedente sessione utente. Grazie alla decostruzione degli oggetti in Kotlin, è possibile ciclare l'intera mappa in appena 2 righe.

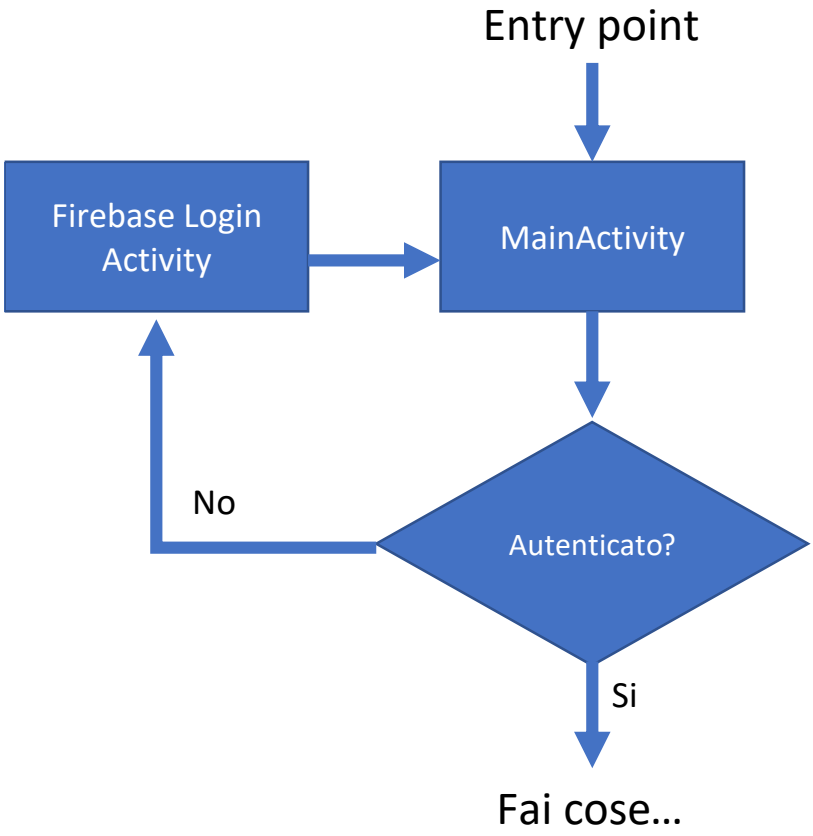
METODO UPLOADIMAGES()

```
private fun uploadImages(files: List<File>, onResult: (paths: List<String>) ->
Unit){
    val paths = ArrayList<String>()
    var i = 0
    if(files.isEmpty()) onResult(paths)
    else for(file in files) {
        val imgName = UUID.randomUUID().toString()
        storage.reference
                .child(PROJECT_IMAGES_PATH).child(imgName)
                .putFile(Uri.fromFile(file)).addOnSuccessListener {
                        paths.add(imgName)
                        i++
                        if(i == files.size) onResult(paths)
                    }
    }
}
```

Questo metodo permette di eseguire l'upload delle foto sul bucket del progetto. Il codice è abbastanza esplicativo, la peculiarità del codice esposto sta nella semplicità d'utilizzo delle APIs di Firebase Storage.

UNO SGUARDO ALL' ONCREATE()

Nonostante possa sembrare controintuitivo, ho scelto di utilizzare una sola Activity in Projector. O per lo meno, ne ho scritta solo una ma ne utilizzo altre (una per il login di FirebaseUI, una per il cropping di immagini). Tutti i cambiamenti di UI e dati da utilizzare verranno gestiti tramite frammenti, più avanti mostrerò come, intanto vediamo il flusso di avvio dell'applicazione:



Il controllo dell'autenticazione avviene all'interno del metodo `onCreate()`:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    if(!Database.isLoggedIn()) {
        initiateLogin()
        return
    }
    …
}
private fun initiateLogin() {
    startActivityForResult(
            AuthUI.getInstance().createSignInIntentBuilder()
                    .setAvailableProviders(Arrays.asList(
                            AuthUI.IdpConfig.EmailBuilder().build(),
                            AuthUI.IdpConfig.GoogleBuilder().build(),
                            AuthUI.IdpConfig.FacebookBuilder().build(),
                            AuthUI.IdpConfig.TwitterBuilder().build()))
                    .build(),RC_SIGN_IN)
}
```

In caso non si è autenticati, `initiateLogin()` costruisce un `Intent` tramite un helper di FirebaseUI e lancia l'Activity di login.

Una volta certi di essere autenticato, costruisco parte della UI e controllo che l'utente corrente abbia tutte le carte in regola per utilizzare l'applicazione senza intoppi:

```
import kotlinx.android.synthetic.main.activity_main.*

override fun onCreate(savedInstanceState: Bundle?) {
        …
        setContentView(R.layout.activity_main)
        setSupportActionBar(toolbar)
        Database.checkCurrentUserIntegrity { isComplete ->
            if (!isComplete) {
                launchFragment(EditProfileFragment::class.java,
        EditProfileFragment.createBundle())
            } else {
                setUpNavigationViewAndGoHome(SetUpNavViewEvent())
            }
}
```

Notate come faccio riferimento a `toolbar` senza aver prima recuperato la view tramite `findViewByld()`. Kotlin offre delle estensioni molto comode per Andorid tra cui le **synthetic properties**. In poche parole, viene tenuta traccia di quale layout è in questo momento mostrato nella nostra activity e per ogni ID dichiarato nell'XML genera un indice degli oggetti istanziati (NB: non dei nome come in `R`!) ed è possibile importarli tramite dei file sintetici generati al volo. Comodo no?

Ma torniamo al nostro `if else`. Se il l'utente ha qualcosa che non va, lanciamo il frammento che si occupa di modificare il profilo utente ma **senza** abilitare la barra di navigazione laterale (torna scomodo se può uscire senza completare il profilo!). Se l'utente corrente è a posto, finisco di costruire la UI e mostro la home. Per ora ignoriamo cosa sia quel parametro `SetUpNavigationEvent()`, che per di più qui è utilizzato solo per comodità.

METODO SETUPNAVIGATIONVIEWANDGOHOME()

Questo metodo permette di completare la configurazione della UI della activity, nel particolare la navigation bar:

```
@Subscribe(threadMode = ThreadMode.MAIN)
fun setUpNavigationViewAndGoHome(event: SetUpNavViewEvent) {
    val toggle = object : ActionBarDrawerToggle(this, drawer_layout,
            toolbar, R.string.navigation_drawer_open,
            R.string.navigation_drawer_close) {

        override fun onDrawerSlide(drawerView: View, slideOffset: Float) {
            EventBus.getDefault().post(CloseKeyBoardEvent())
            super.onDrawerSlide(drawerView, slideOffset)
        }

    }
    drawer_layout.addDrawerListener(toggle)
    toggle.syncState()
    nav_view.setNavigationItemSelectedListener(this)
    launchFragment(AllProjectsFragment::class.java)
    Database.getCurrentUser({
        navigation_display_name.text = it.getDisplayName()
        navigation_email.text = it.email
        if (it.imgUri != null) {
            Database.loadUserProfilePictureToImageView(it,
                    this, navigation_profile_image,
                    {}, {it.printStackTrace()})
        }
        if(it.projectId != null){
            nav_view.menu.getItem(2)
                    .setIcon(R.drawable.ic_folder_shared_black_24dp)
            nav_view.menu.getItem(2)
                    .title = getString(R.string.my_project)
            userProjectID = it.projectId
        } else {
            nav_view.menu.getItem(2)
                    .setIcon(R.drawable.ic_add_black_24dp)
            nav_view.menu.getItem(2).title = getString(R.string.create_project)
            userProjectID = null
        }
        if(it.isAdmin && !adminFlag){
            nav_view.menu.add(R.id.secondary_menu, ADMIN_MENU_ID,
                    0, resources.getString(R.string.admin_menu))
                    .setIcon(R.drawable.ic_supervisor_account_black_24dp)
            adminFlag = true
        }
    }, true)
    Database.getYears {
        if(it!=null){
            ArrayAdapter<String>(this@MainActivity,
                    R.layout.simple_white_spinner_item, it).apply {
                this.setDropDownViewResource(
                        android.R.layout.simple_spinner_dropdown_item)
                toolbar_spinner.adapter = this
                toolbar_spinner.onItemSelectedListener = this@MainActivity
            }
        }
    }
}
```

Questo metodo viene eseguito o in `onCreate()` se l'utente corrente è completo, altrimenti viene eseguito quando l' `EditProfileFragment` , invocato con il suo bundle nel metodo `MainActivity.onCreate()` , pubblica sul bus un `SetUpNavViewEvent` .

Per popolare l'header del drawer, invoco `Database.getCurrentUser( {…}, true)` con il flag `hook = true` , in questo modo ogni volta che l'utente corrente verrà modificato sul RTD, la lambda passata fra parentesi graffe verrà rieseguita, così da aggiornare i dati della UI.
Per evitare che il menù admin appaia più volte in caso di modifica del profilo utente, tengo traccia con un booleano se è già stata aggiunta la voce. Non è elegante ma funziona.
Chiaramente poiché si tratta di modifiche della UI, questo metodo va eseguito sempre nel Main Thread. Se invocato dall' `onCreate()` non è necessario intervenire, ma se invocato tramite bus è necessario specificare su quale thread vada eseguito questo codice specificandolo nell'annotazione.

Nelle ultime righe di codice di questo metodo recupero la lista degli anni scolastici in cui sono stati pubblicati dei progetti dal RTD e popolo lo spinner nella toolbar (lo spinner viene nascosto quando non siamo nell' `AllProjectsFragment` , tale logica viene gestita dal metodo `launchFragment()`).

C'È UNA DOMANDA CHE PUÒ SORGERE SPONTANEA AI PIÙ ATTENTI ED ESPERTI:

Come può il codice all'interno di onCreate() essere eseguito correttamente dal Main Thread al rientro dalla login activity di FirebaseUI?

Fortunatamente l'SDK Android ci aiuta:

```
public override fun onActivityResult(requestCode: Int,
                                     resultCode: Int, data: Intent?) {
    when (requestCode) {
        …

        RC_SIGN_IN -> {
            recreate()
        }
        …
}
```

Il metodo `Activity.recreate()` invalida l'intera activity attuale e la ricostruisce da capo, come fosse appena stata lanciata.

LA NAVIGAZIONE NELL'APP

La navigazione fra le diverse componenti dell'applicazione può avvenire tramite iterazione all'interno di un frammento oppure tramite drawer.

NAVIGAZIONE TRAMITE DRAWER

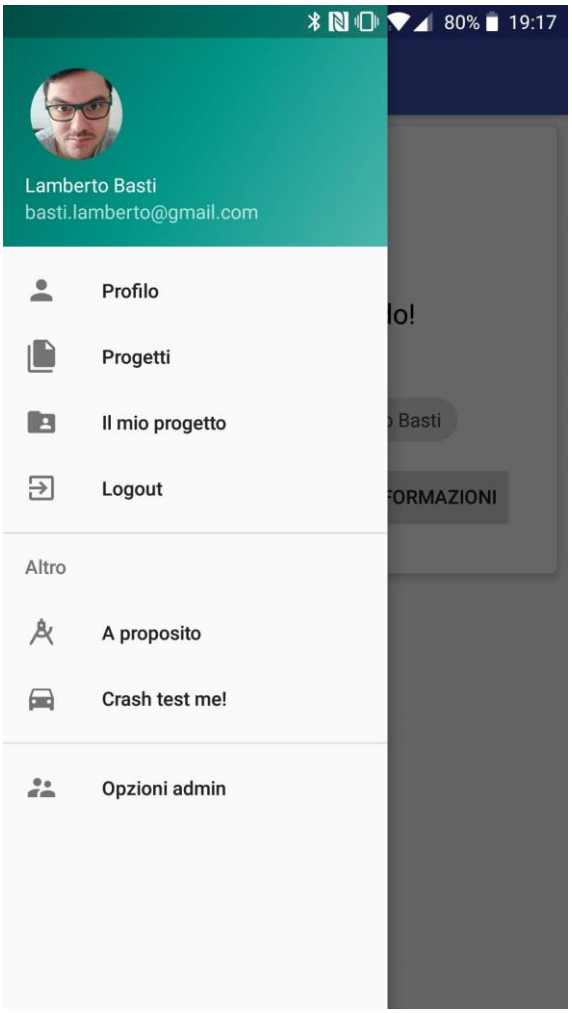
Per intercettare la selezione del menù del drawer, la `MainActivity` estende `NavigationView.OnNavigationItemSelectedListener`:

```
const val ADMIN_MENU_ID = 123123

class MainActivity : AppCompatActivity(),
                NavigationView.OnNavigationItemSelectedListener,
                AdapterView.OnItemSelectedListener {
        ...
      override fun onNavigationItemSelected(item: MenuItem): Boolean {
          // Handle navigation view item clicks here.
          when (item.itemId) {
              R.id.nav_profile -> launchFragment(ProfileFragment::class.java,
                       ProfileFragment
                               .createBundle(Database.getCurrentAuthUserID()))
              R.id.nav_projects -> {
                  launchFragment(AllProjectsFragment::class.java)
              }
              R.id.nav_my_project -> {
                  if(userProjectID != null)
                       launchFragment(ProjectFragment::class.java,
                               ProjectFragment
                                       .createBundle(userProjectID as String))
                  else launchFragment(CreateProjectFragment::class.java)
              }
              R.id.nav_logout -> {
                  val progress = ProgressDialog(this)
                  progress.setTitle(resources.getString(R.string.wait))
                  progress.setMessage(resources.getString(R.string.logging_out))
                  progress.setCancelable(false)
                  progress.show()
                  Database.logOut(this, {
                      progress.dismiss()
                      initiateLogin()
                  })
              }
              R.id.nav_about -> {
                  launchFragment(AboutFragment::class.java)
              }
              ADMIN_MENU_ID -> {
                  launchFragment(AdminPanelFragment::class.java)
              }
              R.id.nav_crash -> {
                  Crashlytics.getInstance().crash()
              }
          }
          drawer_layout.closeDrawer(GravityCompat.START)
          return true
      }
      ...
}
```

Il costrutto `when` rende molto leggibile il codice e permette di comprendere al volo il flusso del codice. In caso di logout viene mostrato un deprecato `ProgressDialog` che mostra un `progressWheel` fino a quando non è terminato il logout.

Più particolare è invece la voce `ADMIN_MENU_ID`. Questa costante definita top level nel file `MainActivity.kt`, viene utilizzata solo quando il proprio account ha il flag admin nel database impostato a `true` perché solo in quel caso viene mostrato il menù da “Opzioni admin” mostrato qui di lato.



NAVIGAZIONE TRAMITE EVENTI

Per navigare tramite eventi ho creato diversi metodi nella `MainActivity` per poter gestire i diversi eventi. Un metodo in particolare merita più attenzione a causa della struttura degli eventi che veicola. Vediamo prima la struttura dell'evento:

```
class FragmentInteractionEvent(val interaction: Interaction){
    abstract class Interaction(var data: String? = null)
    object EditProfile: Interaction()
    object OpenProfile: Interaction()
    object AllProjects: Interaction()
    object OpenProject: Interaction()
    object EditProject: Interaction()
}
```

La classe `FragmentInteractionEvent` ha come unico attributo una `Interaction`, una classe astratta con cui ho creato i 5 `object`s nelle righe successive. In questo modo ho creato un sistema di attributi con tipi personalizzati rendendo type-safe il costruttore di `FragmentInteractionEvent`.

Ora invece vediamo come la `MainActivity` intercetta questi eventi:

```
@Subscribe
fun onFragmentInteraction(event: FragmentInteractionEvent) {
    when(event.interaction){
        FragmentInteractionEvent.EditProfile -> {
            launchFragment(EditProfileFragment::class.java)
        }
        FragmentInteractionEvent.OpenProfile -> {
            launchFragment(ProfileFragment::class.java,
                    ProfileFragment.createBundle(
                            event.interaction.data as String))
        }
        FragmentInteractionEvent.AllProjects -> {
            launchFragment(AllProjectsFragment::class.java)
        }
        FragmentInteractionEvent.OpenProject -> {
            launchFragment(ProjectFragment::class.java,
                    ProjectFragment.createBundle(
                            event.interaction.data as String))
        }
        FragmentInteractionEvent.EditProject -> {
            launchFragment(EditProjectFragment::class.java)
        }
    }
}
```

Ancora una volta la sintassi di Kotlin rende facilmente comprensibile il flusso del codice. In base all'oggetto contenuto dall'evento, viene lanciato un frammento in particolare; se tale frammento necessita di dati aggiuntivi, il rispettivo metodo di `createBundle()` verrà invocato per costruire con argomento il contenuto dell'evento correttamente castato.

CLASS PROJECTORFRAGMENT: FRAGMENT()

Come anticipato prima, tutta la logica dell'applicazione è gestita a frammenti. Per ogni funzionalità è stato creato un frammento che la gestisce e comunica con la `MainActivity` tramite un bus ad eventi. Ora vedremo prima come sono fatti questi frammenti e poi come lanciati.

```
abstract class ProjectorFragment:
                        android.support.v4.app.Fragment() {

    @Suppress("unused")
    @Subscribe
    fun closeKeyboard(event: CloseKeyBoardEvent){
        if(activity?.currentFocus is EditText) {
            Utils.hideKeyboardFrom(activity!!,
                activity!!.currentFocus as EditText)
        }
    }

    override fun onResume() {
        super.onResume()
        EventBus.getDefault().register(this)
    }

    override fun onPause() {
        EventBus.getDefault().unregister(this)
        super.onPause()
    }
}
```

Abbiamo un semplice frammento che si occupa di registrarsi al bus quando inizia ad essere utilizzato e si scollega quando viene chiuso. Il metodo `closeKeyboard()` si occupa semplicemente di chiudere la tastiera quando qualche altro componente pubblica un `CloseKeyBoardEvent`.

Questo frammento è il genitore di tutti i frammenti che gestiscono la UI di Projector. In tutto sono 9, ma insieme ne vedremo solo qualcuno a scopo didattico.

ALLPROJECTSFRAGMENT: PROJECTORFRAGMENT()

```
class AllProjectsFragment: ProjectorFragment() {

    private lateinit var rv: RecyclerView
    private var adapter: ProjectsAdapter? = null

    override fun onCreateView(inflater: LayoutInflater,
                              container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        rv = RecyclerView(this.context)
        rv.layoutParams = CoordinatorLayout
                                .LayoutParams(MATCH_PARENT, MATCH_PARENT)
        rv.layoutManager = LinearLayoutManager(context)
        return rv
    }

    @Subscribe(sticky = true)
    fun bindAdapter(event: BindAdapterInAllProjectsEvent){
        adapter = event.adapter
        rv.adapter = adapter
        adapter!!.startListening()
    }

    override fun onPause() {
        adapter?.stopListening()
        super.onPause()
    }

    override fun onResume() {
        super.onResume()
        adapter?.startListening()
    }
}
```

In questo frammento vengono mostrati i progetti disponibili, inoltre è l'unico frammento in cui costruisco la UI programmaticamente (ovvero senza l'aiuto dell' XML).

Nell' `onCreateView()` inizializzo il `RecyclerView`, unica view utilizzata dalla UI del frammento, gli assegno il necessario per assegnarla al `ViewGroup` genitore e la ritorno al chiamante del metodo.

Il frammento è inoltre in ascolto su `BindAdapterInAllProjectsEvent`s i quali contengono un `ProjectsAdapter` pronto per essere visualizzato nella UI. Gli adapter vengono costruiti dalla `MainActivity` ogni volta che viene scelto un anno scolastico da visualizzare nello spinner in alto:

```
override fun onItemSelected(parent: AdapterView<*>?, view: View?,
                            position: Int, id: Long) {
    val year = (view as TextView).text.toString()
    val adapter = ProjectsAdapter(Database.buildProjectsUiOptions(year), this)
    EventBus.getDefault()
        .removeStickyEvent(BindAdapterInAllProjectsEvent::class.java)
    EventBus.getDefault().postSticky(BindAdapterInAllProjectsEvent(adapter))
}
```

NB: la `MainActivity` estende `AdapterView.OnItemSelectedListener`.

PROFILEFRAGMENT: PROJECTORFRAGMENT()

```
class ProfileFragment: ProjectorFragment() {

    companion object {
        const val TAG = "ProfileFragment"
        fun createBundle(userId: String): Bundle{
            return Bundle().putString_(TAG, userId)
        }
    }

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                              savedInstanceState: Bundle?): View? {
        return container!!.inflate(R.layout.fragment_profile).apply {
            alpha_loading.visibility = View.VISIBLE
            progress_bar.visibility = View.VISIBLE
            val displayedUserId = arguments!!.getString(TAG)
            Database.getUserByUid(displayedUserId,
                      { loadUI(it!!, this)},
                      { Log.e(TAG, it.message)})
            val currentAuthUserId = Database.getCurrentAuthUserID()
            if(currentAuthUserId == displayedUserId){
                edit_profile_button.setOnClickListener {
                    EventBus.getDefault()
                        .post(FragmentInteractionEvent(EditProfile))
                }
                edit_profile_button.visibility = VISIBLE
            } else edit_profile_button.visibility = GONE
            scroll_view.addOnScrollChangedListener { _, Y ->
                if (Y == 0 ) EventBus.getDefault()
                                .post(ChangeToolbarElevationEvent(0f))
                else EventBus.getDefault()
                        .post(ChangeToolbarElevationEvent(8f))
            }
            EventBus.getDefault().post(ChangeToolbarElevationEvent(0f))
        }
    }

    private fun loadUI(it: User, v: View) {
        …
    }
}
```

`ProfileFragment` gestisce la visualizzazione dei profili utente.

Notate il costrutto `companion object`, in Kotlin il modificatore `static` non esiste. Esistono però gli `object` i quali permettono di chiamare dei metodi senza istanziare qualcosa manualmente; un `companion object` non è altro che un `object` con il nome della classe in cui è costruito.
In questo caso, lo utilizzo per costruire un `Bundle` che utilizzerò come `arguments` per il frammento stesso quando verrà lanciato dalla `MainActivity`.

In `onCreateView()` costruisco la UI recuperando i dati tramite `Database.getUserById()` e se l'utente correntemente loggato è lo stesso a cui sto caricando il profilo aggiungo il bottone per modificare il profilo.

Un'altra interessante funzionalità di Kotlin che ho utilizzato è l'extension function `ViewGroup.inflate()`:

```
fun ViewGroup.inflate(layoutRes: Int): View {
    return LayoutInflater.from(context).inflate(layoutRes, this, false)
}
```

Questa semplice funzione aggiunge un metodo alla classe `ViewGroup` che permette di costruire una view più agevolmente.

LIBRERIE UTILIZZATE

Come in ogni software, sono state utilizzate diverse librerie esterne open-source e non. Due di queste sono state create da me appositamente per questo progetto, pubblicate su GitHub e disponibili tramite il repository di **JitPack.io**.

- Google: Chrome Custom Tabs
  https://developer.chrome.com/multidevice/android/customtabs
- Facebook: Facebook Login SDK
  https://developers.facebook.com/docs/facebook-login/android
- Google: Firebase UI
  https://github.com/firebase/FirebaseUI-Android
- Google: Firebase Database
  https://firebase.google.com/
- Google: Firebase Auth
  https://firebase.google.com/
- Google: Firebase Storage
  https://firebase.google.com/
- Bumptech: Glide
  https://github.com/bumptech/glide
- Nex3z: Flow Layout
  https://github.com/nex3z/FlowLayout
- Robert Levonyan: Material Chip View
  https://github.com/robertlevonyan/materialChipView
- ArthurHub: Android Image Cropper
  https://github.com/ArthurHub/Android-Image-Cropper
- Twitter: Twitter core SDK
  https://github.com/twitter/twitter-kit-android
- Yarolegovich: Discrete ScrollView
  https://github.com/yarolegovich/DiscreteScrollView
- Hdodenhof: Circle ImageView
  https://github.com/hdodenhof/CircleImageView
- Zeibaitsu: Compressor
  https://github.com/zetbaitsu/Compressor
- GreenRobot: EventBus
  http://greenrobot.org/eventbus/
- Lamba92: ImagesAdapter
  https://github.com/lamba92/imagesAdapter
- Lamba92: Utils
  https://github.com/lamba92/Utils