

Personal Health Monitor

MONITORA, GESTISCI E CONTROLLA LA TUA SALUTE

Descrizione

L'applicazione Android Personal Health Monitor (PHM) ha lo scopo di aiutare l'utente a gestire in modo autonomo, controllare e monitorare a suo piacimento, dei parametri considerati rilevanti per la sua salute. PHM è in grado di tracciare diversi tipi di grafici al fine di avere un impatto visivo sull'andamento dei valori che si desidera monitorare. I report così creati dall'utente possono essere filtrati in base a diversi filtri di ricerca al fine di aiutare lo user a reperire rapidamente e intuitivamente le informazioni che sta cercando.

Componenti

PHM è stata implementata con Android Studio tramite linguaggio Java. Come template iniziale è stato usato un Navigation Drawer Activity, che di default costruisce una sezione a scorrimento (il “drawer” appunto) tramite il quale si può accedere alle diverse parti dell'applicazione. La SDK minima richiesta è la API 16: Android 4.1 (Jelly Bean) che compila approssimativamente nel 99,8% dei dispositivi.

L'applicazione si compone di 4 parti principali: la sezione dei **Report**, la sezione di **Ricerca**, la sezione dei **Grafici** e le **Impostazioni**, tutte accessibili tramite l'apertura del drawer, a cui è possibile accedere tramite swipe da sinistra verso destra. Tutte e 4 le sezioni sono Fragment implementati dall'IDE. L'applicazione è personalizzabile e facile da utilizzare, volutamente spoglia di troppi dettagli e/o pulsanti affinché il suo utilizzo risulti facilitato anche da chi non è pratico o non è un utilizzatore assiduo di dispositivi Android.

Report

All'avvio del PHM la prima sezione che si apre è quella del report. Ci sono 6 elementi di tipo CardView accoppiati a due per due in modo da riempire lo schermo. Lo sfondo utilizzato è stato fatto tramite un file xml utilizzato per creare l'effetto sfumatura. Il percorso del file in questione si trova in res/drawable/gradient_background.xml e sarà lo sfondo che accompagnerà l'utente attraverso tutta l'applicazione.

Le CardView hanno un colore associato a ciascuna di esse, che riflette l'importanza che quella particolare informazione possiede per l'utente. Ciascuna CardView ha una icona rappresentativa dell'informazione, il nome dell'informazione monitorata e una rappresentazione

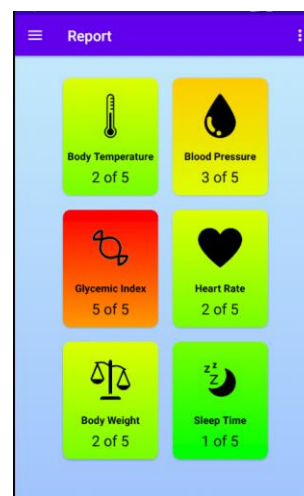
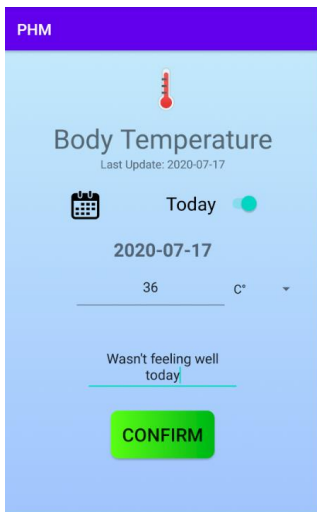


Figura 1 Le CardView del Report

numerica dell'importanza che può assumere (da 1 a 5). Le importanze al primo accesso sono impostate di default, ma sono completamente modificabili.

Le informazioni monitorabili sono: temperatura corporea, pressione sanguigna, indice glicemico, battito cardiaco, peso corporeo e ore di sonno.

Al tocco di una qualsiasi di queste CardView appare un form da compilare con le relative informazioni. Ci sarà, come sempre, un'icona rappresentativa seguita dal nome dell'informazione. Al di sotto del nome dell'informazione al primo accesso apparirà la scritta *"No records inserted"* poiché non è stata inserita alcun valore. Al di sotto è presente l'icona cliccabile di un calendario. Al tocco da parte dell'utente si aprirà un DatePickerDialog, in cui egli potrà scegliere a piacimento una data in cui inserire l'informazione. È stata incontrata una difficoltà nel settare il testo della TextView che doveva contenere la data scelta poiché era sfuggito che la numerazione dei mesi partiva da zero e non da uno come erroneamente supposto. La data scelta veniva poi trasformata in un formato uguale per tutte le sezioni della app per semplicità implementativa e segue il formato americano: "yyyy-MM-dd". Se l'utente invece vuole inserire direttamente la data odierna senza dover aprire il Dialog, semplicemente deve attivare lo Switch alla destra dell'icona del calendario. Questo switch quando attivo cambia il testo della TextView e le assegna la data odierna.



2 Inserimento della temperatura corporea

Sotto la data ci sarà una sezione per l'input dell'utente, diversa a seconda dell'informazione che si vuole immettere. Per quanto riguarda Body Temperature, Heart Rate, Body Weight e Sleep Time è presente un semplice EditText, con inputType di tipo *number*; invece per Blood Pressure vi sarà una SeekBar da cui poter scegliere un range di valori che vanno da 0 a 300mmHg; per Glycemic Index invece sarà presente sempre una SeekBar ma con solo 4 valori possibili. Nel caso di Body Temperature e Body Weight sarà inoltre possibile anche scegliere l'unità di misura, gradi Celsius o Fahrenheit per la temperatura, e Kilogrammi o Libbre per il peso. Qualsiasi sia l'unità di misura scelta i dati verranno convertiti e inviati in gradi Celsius per la temperatura e in Kilogrammi per il peso corporeo, per una questione di facilità di utilizzo e di recupero dati dal Database. Vi è un ulteriore vincolo per

quando riguarda le ore di sonno: non è possibile naturalmente inserire un quantitativo di ore di sonno superiore a 24, e l'utente verrà in questo caso avvertito che i dati di input non sono validi.

Per ogni CardView sarà presente un EditText con hint “*Optional Note*” in cui l’utente a sua discrezione potrà inserire o meno una nota opzionale. Come valore di default al Database verrà inviato *null*.

Dopo il primo inserimento, nel sottotitolo dove prima si leggeva “*No record inserted*” apparirà invece la scritta “*Last update: yyyy-MM-dd*” con la data dell’ultimo inserimento eseguito. Affinché un inserimento possa essere considerato tale è necessario premere il tasto “*CONFIRM*” per inviare i dati al database. Se l’inserimento avviene con successo apparirà un Toast con la conferma dei dati salvati, in caso contrario l’utente verrà avvertito che non è stato possibile inserire i dati. Quest’ultimo caso si presenta quando l’utente non inserisce una data, quando la data non è valida (ad esempio una data nel futuro) oppure quando non inserisce l’informazione per quella particolare CardView.

Search

In questo secondo Fragment viene presentata all’utente la possibilità di cercare in base a dei filtri i report o le informazioni di cui necessita, in modo rapido e intuitivo. Al tocco su “*Search*” all’utente apparirà una schermata con un titolo e 3 CardView. Egli potrà scegliere se visualizzare i dati in base al tipo di informazione (“*By information*”), in base alla data (“*By date*”), o in base al criterio di importanza (“*By importance*”).

- **By information:** cliccando su questa opzione all’utente apparirà una ListView con i 6 tipi di informazione, con relativa immagine e livello di importanza rappresentato tramite RatingBar a 5 stelle (l’implementazione di questa lista è nel file `ListData.java`). Cliccando su una qualsiasi delle informazioni si aprirà una ulteriore ListView a scorrimento (`ViewListData.java`), che avrà un *header* contenente il nome dell’informazione e tutti i dati inseriti dall’utente per quel particolare parametro, con annessi il giorno dell’inserimento, il valore dell’informazione e l’eventuale nota opzionale. Per ogni Item della ListView ci saranno 2 Button: *Modify* e *Delete*. Al tocco del bottone per la modifica si aprirà un

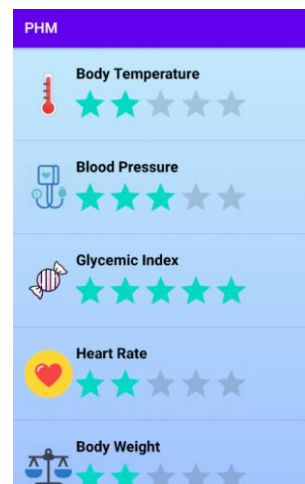


Figura 3 Ricerca per tipo di informazione

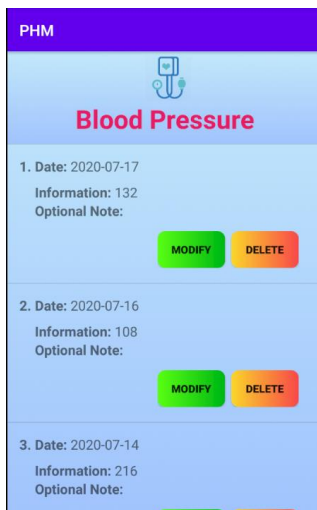


Figura 4 ListView della Blood Pressure

Dialog classico con la possibilità di modificare tutti i valori di quel particolare Item. Si dovrà dunque confermare le modifiche tramite “SAVE”, o chiudere la finestra di Dialog tramite “CANCEL”. Il Button DELETE invece aprirà un Dialog chiedendo la conferma per l’eliminazione di quell’Item dalla lista e dunque dal Database. Una volta premuto DELETE l’operazione è irreversibile. Per entrambi le operazioni di modifica o eliminazione è necessario ricaricare la ListView per vedere gli effetti dell’operazione.

L’implementazione della ListView ha comportato qualche difficoltà ma solo a livello concettuale, poiché è necessario avere un file layout dedicato solo alla ListView e un altro dedicato solo alla singola riga della ListView.

Una volta capito questo concetto, la ListView si è rivelata molto robusta e versatile, motivo per cui è stata utilizzata

ampiamente all’interno di PHM. Ciascuna ListView è formata da due parti: una classe List per la costruzione della lista e una classe Adapter per l’implementazione della singola riga della lista.

- By date:** scegliendo questa opzione, l’utente può visualizzare i dati inseriti per una certa informazione in un particolare giorno. Più precisamente, nel caso in cui vi siano più inserimenti della stessa informazione in un solo giorno, l’utente potrà visualizzare in questa sezione l’andamento dei valori attraverso tutto l’arco della giornata. Al tocco della CardView si aprirà una schermata con un calendario, da cui l’utente dovrà selezionare una data. Anche qui se l’utente sceglie una data futura, apparirà un Toast che avverte dell’impossibilità di tale avvenimento. La data scelta, se valida, apparirà in basso a destra in una TextView. A sinistra della TextView ci sarà uno Spinner, dove l’utente potrà selezionare una tra le 6 informazioni possibili per poi cliccare NEXT e accedere ai suoi dati. Nella schermata successiva si aprirà una ListView con un *header* indicante l’informazione scelta e la data, seguita da una media in Float dei valori inseriti per quel particolare giorno. La row di ciascuna lista sarà composta da “Record No. N” che indica l’ordine cronologico di inserimento dell’informazione in quel giorno, dal valore dell’informazione e

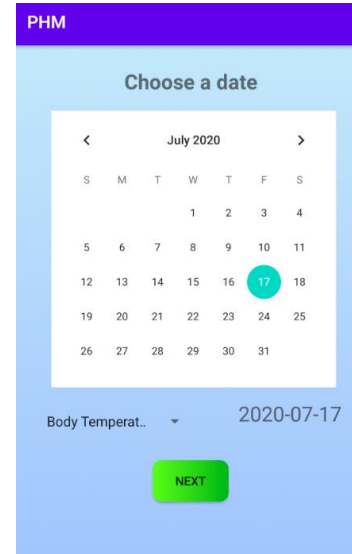


Figura 5 Ricerca per data

dalla nota opzionale. L'implementazione di questa lista è nel file `ListDataOnDate.java`.

- **By importance:** cliccando sull'ultima opzione, all'utente si apre la possibilità di visualizzare i report che considera importanti. La schermata infatti permette all'utente di selezionare una data (non nel futuro) tramite `DatePickerDialog`, e di scegliere tramite `SeekBar` il livello minimo di importanza di una informazione affinché venga visualizzata nella schermata successiva. Per esempio se l'utente vuole visualizzare i report con importanza superiore a 3 dal 1 luglio 2020 in poi, sceglierà la data dal calendario, muoverà la `SeekBar` al livello 3 e premerà NEXT. Come si può intuire, grazie all'utilizzo di questi filtri, risulta molto facile cercare un report di cui si è persa la traccia nel tempo o di cui ci si è dimenticati e si vuole consultare.

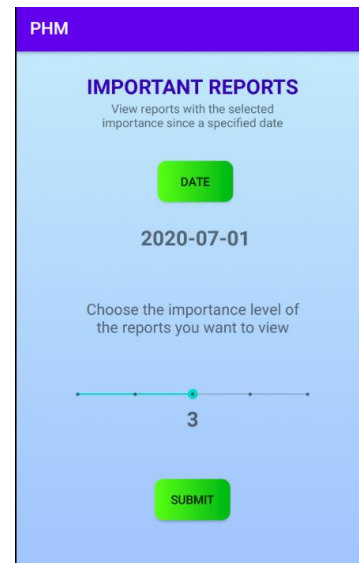


Figura 6 Ricerca per importanza

PHM	
Information:	Blood Pressure
Since:	2020-07-01
Value:	132
Date:	2020-07-17
Value:	108
Date:	2020-07-16
Value:	216
Date:	2020-07-14
Value:	165

Alla schermata successiva vengono visualizzate le informazioni con dati inseriti che rispettano i filtri appena impostati. Apparirà una `ListView` con nome dell'informazione e il suo livello di importanza. Il codice di questa `ListView` è nel file `ListReport.java`, con annesso relativo `Adapter` chiamato `ImportantTablesAdapter`. Come per il caso precedente, questa `ListView` porta ad un'altra `ListView` (`ViewListReport.java`) con gli effettivi dati richiesti: sarà presente sempre l'*header* con i filtri applicati e sotto la dicitura "*Value*" con il valore, e "*Date*" con la data di inserimento.

Figura 7 Visualizza i dati per importanza

Graphs

Se si clicca sull'opzione "Graphs" si approda nel relativo Fragment. È data all'utente la possibilità di scegliere tra quattro tipi di grafici per visualizzare i suoi dati. Queste 4 possibilità sono presentate tramite CardView dove oltre al nome vi è anche una piccola immagine rappresentante il grafico in questione. I grafici sono stati fatti sfruttando la libreria open source MPAndroidChart Versione 3.1.0

(<https://github.com/PhilJay/MPAndroidChart>) ad opera di Philip Jahoda. È bastato aggiungere le dependencies e la repository maven al file build.gradle. È stata scelta questa libreria poiché è la più popolare e anche la più facile da implementare. I grafici sono altamente customizzabili, ma in PHM come detto in incipit, per questioni di scelte strategiche, il tutto è stato reso il più leggibile e intuitivo possibile, senza l'aggiunta di caratteristiche che avrebbero reso i grafici eccessivamente pesanti a chi non è un *abituè*.

Il layout per i 4 tipi di grafici è lo stesso: una TextView che chiede di scegliere il tipo di informazione che si vuole visualizzare, affiancato da uno Spinner, con la possibilità di scegliere uno tra i 6 possibili parametri. Al di sotto, una TextView che chiede di scegliere un range di date da cui partire per fare l'osservazione sui grafici. Le possibilità sono 6: ultimi 5 giorni, l'ultima settimana, le ultime 2 settimane, l'ultimo mese o tutti i dati. L'utente dopo aver impostato i filtri clicca sul Button "SHOW CHART". Verrà costruita tramite una animazione di 2000 millisecondi, il grafico richiesto.

- **BAR CHART:** l'implementazione dell'istogramma è stato quello che ha portato maggiori difficoltà, la cui principale risiedeva nel trasformare le date (stringhe) in valori utilizzabili come asse x del grafico, cosa che non si è verificata in tutti i

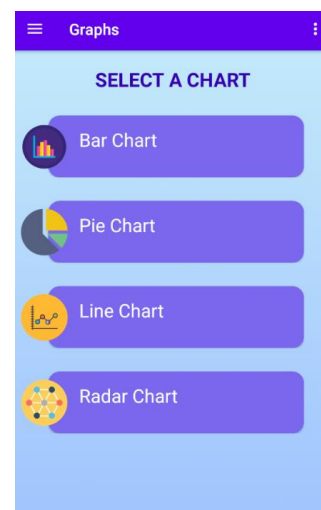


Figura 8 Scelta dei grafici



Figura 9 BarChart per la Body Temperature

grafici. È stato necessario implementare una classe `GetInfo.java` che permettesse di recuperare le informazioni per poi aggiungerle ad un `arraylist` che verrà poi utilizzato per popolare l'asse x dell'istogramma. Un'altra difficoltà parzialmente risolta sono le date che appaiono sopra ciascuna barra dell'istogramma. Esse venivano tagliate nonostante il margine tra i due layout venisse incrementato, per cui anziché visualizzare l'intera data, vengono visualizzati solo il giorno e il mese. Un altro grattacapo che si è dovuto affrontare è stato il nome del file da attribuire a questa sezione. Dopo svariate ricerche in rete ci si è accorti che il nome del file non potesse essere `BarChart.java` per questioni di interferenze con la libreria. Si è deciso così di aggiungere un prefisso "My" al nome del file di ogni grafico, e ciò ha risolto il problema.

- **PIE CHART:** l'implementazione di `PieChart` non ha portato particolari problemi. Si recuperano i dati dal database secondo i filtri impostati dall'utente e li si inseriscono negli `arraylist` *dates* e *information*. Si popola quindi un `arraylist` *value* di tipo `<PieEntry>` con i due `arraylist` appena ottenuti e si costruisce il dataset del grafico tramite



```
"PieDataSet pieDataSet = new PieDataSet(value, "months")"
```

dove "months" è la label da dare allo spicchio della torta. Il dataset appena ottenuto si passa quindi all'oggetto `PieData` tramite i seguenti comandi:

```
<< PieData pieData = new PieData(pieDataSet);
    PieChart pieChart = findViewById(R.id.pieChart);
    pieChart.setData(pieData); >>
```

Figura 10 Pie Chart della Body Temperature

A piacimento si possono poi aggiungere varie feature al grafico ad esempio l'animazione a 2000 millisecondi, o i colori. Ovviamente essendo un grafico a torta i valori vengono trasformati in percentuali, per cui l'utente non troverà il valore effettivamente inserito nella sezione Report ma la percentuale di quel valore in relazione agli altri, nel periodo selezionato.

- **LINE CHART:** la line chart segue la stessa implementazione del BarChart. L'utente sceglie i filtri, sulla base del tipo di informazione e sul periodo di tempo che vuole prendere in considerazione. Le date recuperate vengono trasformate e inserite in un arraylist sempre grazie alla classe GetInfo.java, che vengono poi elaborate e utilizzate per costruire il grafico a linea continua. Per avere un impatto visivo, ciascuna barra indicante un giorno, viene colorata, così da distinguerla da giorni precedenti o successivi. Si può customizzare gran parte del grafico, di seguito alcuni valori scelti:

- `setCircleRadius(10)`
- `setLineWidth(15)`
- `setValueTextSize(15)`
- `setColors(ColorTemplate.MATERIAL_COLORS)`



Figura 11 Line Chart della Body Temperature

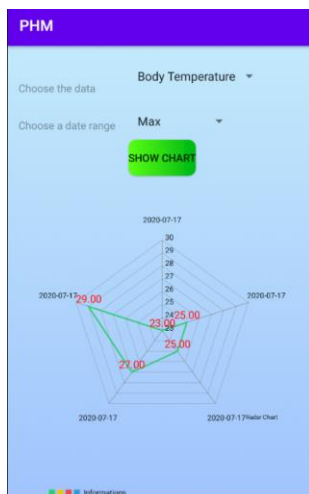


Figura 12 Radar Chart della Body Temperature

- **RADAR CHART:** l'ultimo tipo di grafico implementato è il Radar Chart, che tra quelli disponibili nella libreria MPAndroidChart è parso quello più adatto. Il radar chart costruisce un grafico a tela di ragno dove i valori più alti sono quelli più esterni e quelli più piccoli collasano verso l'interno. Dopo la scelta sugli Spinner e dopo il click sul pulsante viene creata, attraverso animazione, la struttura del grafico con una linea spezzata che segue i vari livelli della "tela". Come per tutti i grafici, è stato necessario implementare un `AdapterView.OnItemClickListener` con i suoi due metodi annessi `onItemSelected` e `onNothingSelected` per permettere la scelta sugli Spinner.

Settings

Il Fragment SettingsFragment è composto di 3 parti principali: “*Set Reminder Time*”, “*Set Importance*” e “*Thresholds*”. Al tocco sulla prima opzione apparirà un TimePickerDialog dove l’utente potrà selezionare un orario per il suo *daily reminder*. Ciò significa che se l’utente non ha inserito nessuna informazione in nessuno dei parametri per la giornata odierna, scatterà una notifica (implementata tramite AlarmManager) all’orario indicato in questa fase. Perché l’orario venga letto dal sistema è necessario cliccare il Button “SAVE”. In caso ci sia stato anche un solo inserimento per qualsiasi parametro per la data odierna, la notifica non scatterà. È possibile inoltre attivare la vibrazione alla notifica tramite uno Switch, che di default rimarrà su “OFF”. La gestione della notifica viene affidata alla classe NotificationService.java. In questa classe, vengono fatte le query al database per verificare la presenza di report per il giorno odierno, e viene inizializzato l’intent passandogli come parametri il context e la MainActivity.java :

```
Intent myIntent = new Intent(context, MainActivity.class);
```

Cosicché all’utente, quando apparirà la notifica, basterà pigiare su di essa per aprire direttamente il fragment per l’inserimento dei dati per il suo report giornaliero. Per fare ciò, è necessario che la API sia la 26 o superiore, poiché il NotificationCompat.Builder richiede come parametri anche il CHANNEL_ID oltre che al context:

```
NotificationCompat.Builder builder = new  
NotificationCompat.Builder(context, CHANNEL_ID);
```

La seconda parte delle impostazioni è il Button “*Set Importance*”: una volta cliccato si aprirà una schermata con la lista delle varie informazioni affiancate da uno Spinner indicante la sua importanza. L’importanza inizialmente avrà dei valori che vengono preimpostati da PHM, ma che sono completamente modificabili. Si parte dal valore più basso, 1, fino al più alto 5, e a seconda del valore scelto cambieranno tutte le parti che coinvolgono l’importanza dell’informazione, ad esempio il colore della CardView nel ReportFragment, o il RatingBar nella ListView filtrata per informazione. Per confermare qualsiasi cambiamento è necessario pigiare il Button “SAVE”, e automaticamente verranno inviate le informazioni al database.

PHM	
Choose the level of importance for your information	
1: Lowest - 5: Highest	
Body Temperature	2 ▾
Blood Pressure	3 ▾
Body Weight	2 ▾
Glycemic Index	5 ▾
Heart Rate	2 ▾
Sleep Time	1 ▾
<div>SAVE</div>	

Figura 13 Schermata “Set Importance”

L'ultima parte delle Impostazioni riguarda i *Thresholds*. L'utente può scegliere quali informazioni monitorare con più cura, e quale deve essere il "limite" da non raggiungere per un prestabilito periodo di tempo. Più precisamente l'utente sceglie un periodo di tempo tramite uno Spinner, le cui scelte possibili sono : "5 days", "1 week", "2 weeks", "3 weeks", "1 month", "2 months". In seguito l'utente può inserire tramite tastiera in un EditText, un valore che ritiene soglia per ogni informazione. Se viene attivato lo Switch, allora il sistema calcolerà automaticamente la media dei valori nell'ultimo periodo prestabilito. Se tale media supera la soglia, verrà inviata una notifica indicante quale parametro è da monitorare con maggior attenzione poiché si è raggiunto il threshold. È da notare che alcuni Switch all'apertura della schermata saranno già su ON. Questo perché PHM automaticamente setta le notifiche per quelle informazioni con importanza maggiore o uguale a 3. Una volta modificate le impostazioni a proprio piacimento, si pigia sul Button SAVE per salvare le informazioni e poter ricevere le notifiche e salvare i valori soglia.

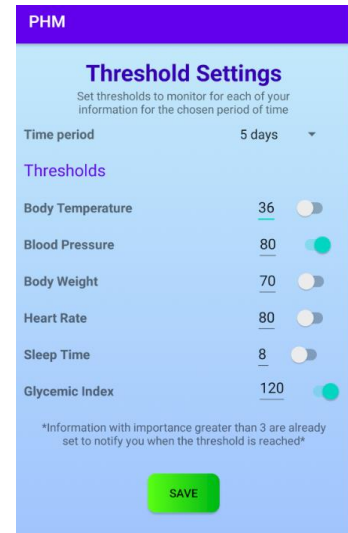


Figura 14 Impostazione dei valori soglia e notifiche

Database

Le operazioni che interagiscono con il database sono state tutte implementate in una classe unica chiamata *DatabaseHelper.java* che estende *SQLiteOpenHelper*. Il nome del database è "**my_health.db**", e sono state create diverse tabelle rese più omogenee possibili per facilitare l'interazione tra di esse. Ad esempio le tabelle "**body_temp_table**", "**blood_pressure_table**", "**glycemic_index_table**", "**heart_rate_table**", "**body_weight_table**", e "**sleep_time_table**" possiedono tutte le stesse colonne: "date", "info" e "optionalNote". Le tabelle della Body Temperature e Body Weight possiedono in aggiunta la colonna "scale" per l'unità di misura.

Nella creazione delle tabelle è stato impostata una colonna "ID" come chiave primaria auto-incrementante, la colonna della data è stata impostata di tipo DATE, e la colonna della nota opzionale è stata impostata di default a *null*.

```
public void onCreate(SQLiteDatabase db) {
    //BODY TEMP TABLE CREATION QUERY
    String bodyTempCreation = "CREATE TABLE " + BODY_TEMP_TABLE + "(ID INTEGER
    PRIMARY KEY AUTOINCREMENT,"
        + dateCol + " DATE, "
        + infoCol + " TEXT, "
        + tempScaleCol + " TEXT, "
```

```
+ optionalNoteCol + " TEXT DEFAULT NULL)";  
db.execSQL(bodyTempCreation);
```

Questo schema è stato seguito per la creazione di tutte le tabelle.

Per quanto riguarda gli inserimenti invece, sono stati creati metodi booleani appositi per ogni tipo di tabella, anche se molto simili è stato ritenuto che fosse comunque meglio separarli, al fine anche di una leggibilità maggiore del codice. Come nella creazione delle tabelle, gli inserimenti seguono tutti una struttura simile per quanto riguarda l'implementazione. Il risultato della *insert* ritorna -1 in caso non sia andata a buon fine.

```
public boolean insertBodyTemp(String date, String temp, String scale, String optionalNote) {  
    SQLiteDatabase db = this.getWritableDatabase();  
    ContentValues contentValues = new ContentValues();  
    contentValues.put(dateCol, date);  
    contentValues.put(infoCol, temp);  
    contentValues.put(tempScaleCol, scale);  
    contentValues.put(optionalNoteCol, optionalNote);  
    long result = db.insert(BODY_TEMP_TABLE, null, contentValues);  
    if (result == -1)  
        return false;  
    else  
        return true;  
}
```

Sono stati implementati 4 metodi separati per ottenere separatamente le 4 colonne fondamentali di ciascuna tabella: *getIds()*, *getDates()*, *getInfo()*, *getOptionalNote()*. Ciascuno di questi metodi richiede che sia passata una stringa come parametro. Questa stringa deve essere la query che verrà poi effettuata al database. Tramite cursore poi, si itera il risultato della query e la si aggiunge ad un arraylist di tipo stringa, che è anche il tipo di ritorno dei metodi appena citati.

```
public ArrayList<String> getIds(String query) {  
    SQLiteDatabase db = this.getReadableDatabase();  
    ArrayList<String> arrayList = new ArrayList<>();  
    try {  
        Cursor cursor = db.rawQuery(query, null);  
        while (cursor.moveToNext()) {  
            arrayList.add(cursor.getString(0) + ". ");  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    db.close();  
    return arrayList;  
}
```

```

public ArrayList<String> getDates(String query) {
    SQLiteDatabase db = this.getReadableDatabase();
    ArrayList<String> arrayList = new ArrayList<>();
    try {
        Cursor cursor = db.rawQuery(query, null);
        while (cursor.moveToNext()) {
            arrayList.add(cursor.getString(cursor.getColumnIndex("date")));
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    db.close();
    return arrayList;
}

public ArrayList<String> getInfo(String query) {
    SQLiteDatabase db = this.getReadableDatabase();
    ArrayList<String> arrayList = new ArrayList<>();
    try {
        Cursor cursor = db.rawQuery(query, null);
        while (cursor.moveToNext()) {
            arrayList.add(cursor.getString(cursor.getColumnIndex("info")));
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    db.close();
    return arrayList;
}

public ArrayList<String> getOptionalNote(String query) {
    SQLiteDatabase db = this.getReadableDatabase();
    ArrayList<String> arrayList = new ArrayList<>();
    try {
        Cursor cursor = db.rawQuery(query, null);
        while (cursor.moveToNext()) {
            arrayList.add(cursor.getString(cursor.getColumnIndex("optionalNote")));
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    db.close();
    return arrayList;
}

```

Si è resa necessaria anche l'implementazione di alcuni metodi specifici, quali `getAvgInfo()` per ottenere la media dei valori, o `countRows()` per ottenere il numero di righe di una particolare query.

È stata inoltre creata un'ulteriore tabella “*importance_table*”, con le colonne “*info_name*” e “*importanceLevel*”, per mantenere salvate le informazioni riguardanti

i livelli di importanza delle varie informazioni. In questo caso, oltre alla creazione venivano aggiunti dei valori iniziali alle tabelle.

```
//IMPORTANCE TABLE
String importance = "CREATE TABLE " + IMPORTANCE_TABLE + "(ID INTEGER
PRIMARY KEY AUTOINCREMENT, "
    + info_name + " TEXT, "
    + importanceLevel + " TEXT)";
String populate = "REPLACE INTO " + IMPORTANCE_TABLE + "
VALUES(1,'body_temperature_table','2'), " +
    "(2,'blood_pressure_table','3'), " +
    "(3,'glycemic_index_table','5'), " +
    "(4,'sleep_time_table','1'), " +
    "(5,'body_weight_table','2'), " +
    "(6,'heart_rate_table','2')";
db.execSQL(importance);
db.execSQL(populate);
}
```

Altri metodi che è importante citare sono `updateRec()`, per eseguire l'update quando l'utente vuole modificare una informazione già inserita

```
public boolean updateRec(String id, String date, String info, String
optionalNote, String table_name) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues contentValues = new ContentValues();
    contentValues.put("date", date);
    contentValues.put("info", info);
    contentValues.put("optionalNote", optionalNote);
    if (!isThisDateValid(date)) {
        return false;
    } else {
        db.update(table_name, contentValues, "id = ? ", new String[]{id});
        return true;
    }
}
```

dove il metodo `isThisDateValid()` verifica la validità nel formato di una data inserita.

References:

Per la costruzione dei grafici:

<https://github.com/PhilJay/MPAndroidChart>

Video/Tutorial

https://www.youtube.com/channel/UCXye_XpNiGkCPwQabGUEqiQ

https://www.youtube.com/channel/UC_Fh8kvtkVPkeihBs42jGcA

<https://www.youtube.com/user/ProgrammingKnowledge>