EasyVersity

Tomassini Danilo
Cappella Simone
Mannini Luca

Ingegneria Informatica e dell'Automazione

Settembre, 2019

Indice

1 Obbiettivi **3**

2 Funzionalità **3**
2.1 Orario . . . 4
2.2 I miei appunti . . . 4
2.3 Appunti condivisi . . . 5
2.4 Impostazioni . . . 6

3 Implementazioni Android studio **7**
3.1 Classe SupportTask . . . 7
3.2 Login e Registrazione . . . 8
3.3 Orario . . . 10
3.3.1 Gestione orario . . . 11
3.4 I miei appunti . . . 17
3.4.1 Aggiungi una materia . . . 17
3.4.2 Vedi appunti . . . 18
3.5 Appunti condivisi . . . 22
3.6 Impostazioni . . . 22
3.6.1 Modifica Username . . . 22
3.6.2 Modifica Password . . . 24
3.7 Database . . . 24
3.7.1 DataAppLoc . . . 24
3.7.2 DataManager . . . 26

4 Implementazioni Xamarin **28**
4.1 Login e Registrazione . . . 28
4.2 Orario . . . 28
4.3 I miei appunti . . . 28
4.4 Appunti condivisi . . . 30
4.5 Impostazioni . . . 30
4.6 Database . . . 30

1 Obbiettivi

EasyVersity rappresenta uno strumento di supporto per lo studente.
Permette di gestire:

- **Orario:** salva l'orario delle lezioni nella tua applicazione per consultarlo quando vuoi.
- **Archivio appunti locale:** da la possibilità di salvare appunti raggruppandoli per materia, indicando titolo e data si può contestualizzare al meglio l'appunto in questione.
- **Condivisione appunti:** rende possibile la condivisione ed il download degli appunti.
- **Impostazioni:** da qui si possono cambiare informazioni come username e password, eprendere visione di info "about us".

2 Funzionalità

Avviata l'applicazione ci si trova davani alla **First activity**; qui si può scegliere di effettuare il login o registrarsi. Una volta effettuato il login si può accedere al menù principale che prevede 4 scelte:

- **Orario.**
- **I miei appunti.**
- **Appunti condivisi.**
- **Impostazioni.**

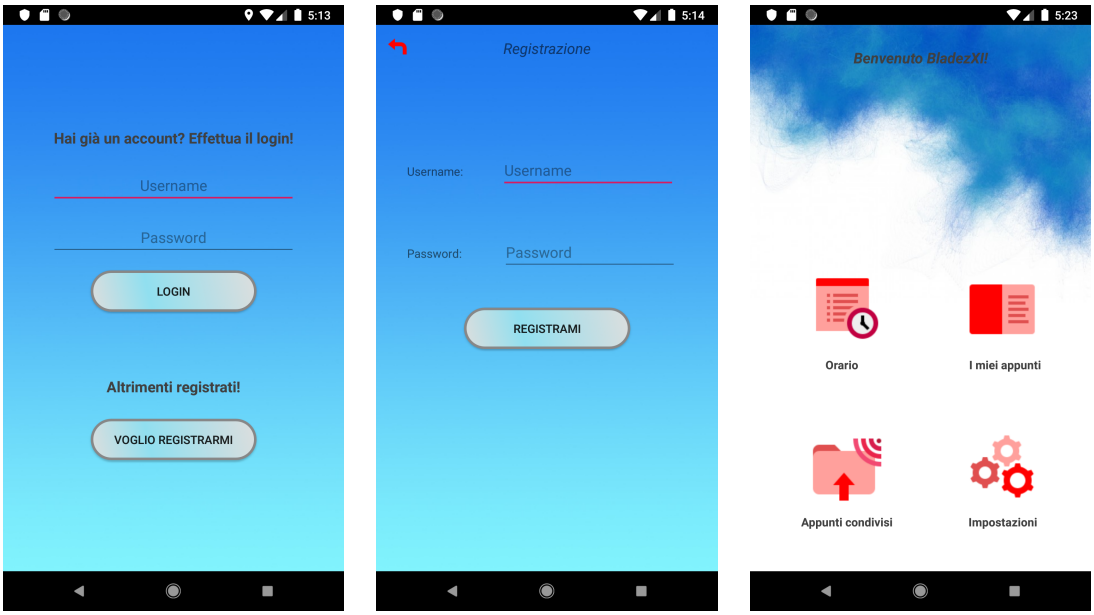

Figura 2.1: Schermata Login, Registrazione e Menu principale.

2.1 Orario

Aperto l'orario ci si trova di fronte alla tabella relativa al giorno corrente; con la bottom navbar si può scegliere il giorno su cui andare ad agire. Premendo nel campo **materia**, relativo all'ora di interesse si può inserire la materia in tabella, si accede, infatti, ad una lista predefinita di materie (questa può essere estesa scegliendo di inserirne una manualmente con il bottone "altro"). Scelta la materia viene visualizzata una finestra di dialogo che permette di inserire l'aula in cui si svolgerà la lezione e la durata della stessa.

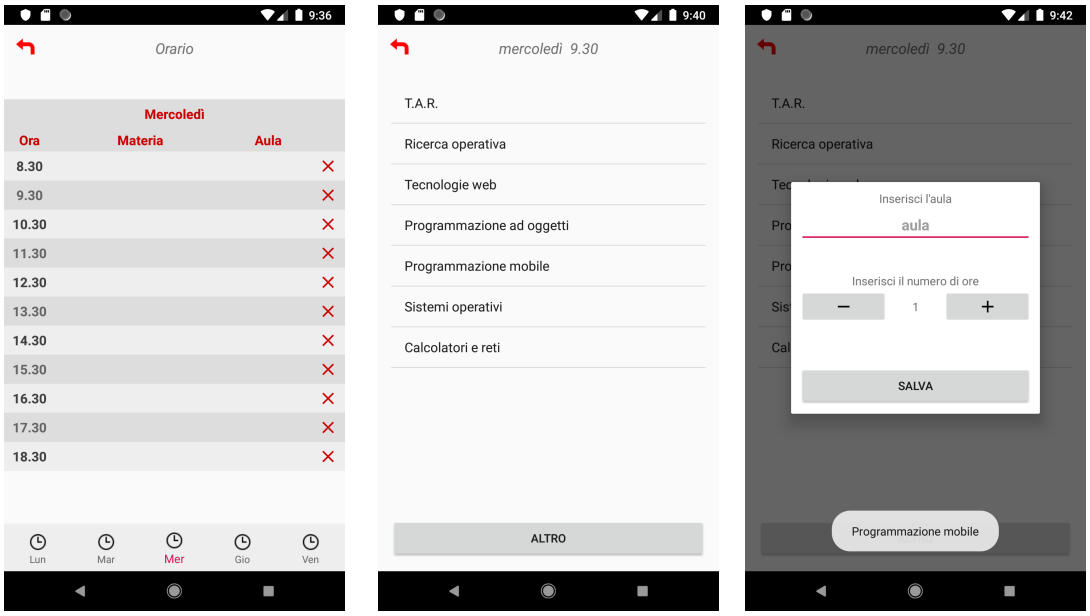

Figura 2.2: Orario.

Salvata la materia nell'orario questa viene salvata nel database e inserita in una lista nella sezione **i miei appunti**.

2.2 I miei appunti

In questa sezione abbiamo accesso alla lista di tutte le materie salvate nel database. Abbiamo la possibilità di aggiungerne altre a piacimento o eliminarle inserendo il nome esatto della materia.

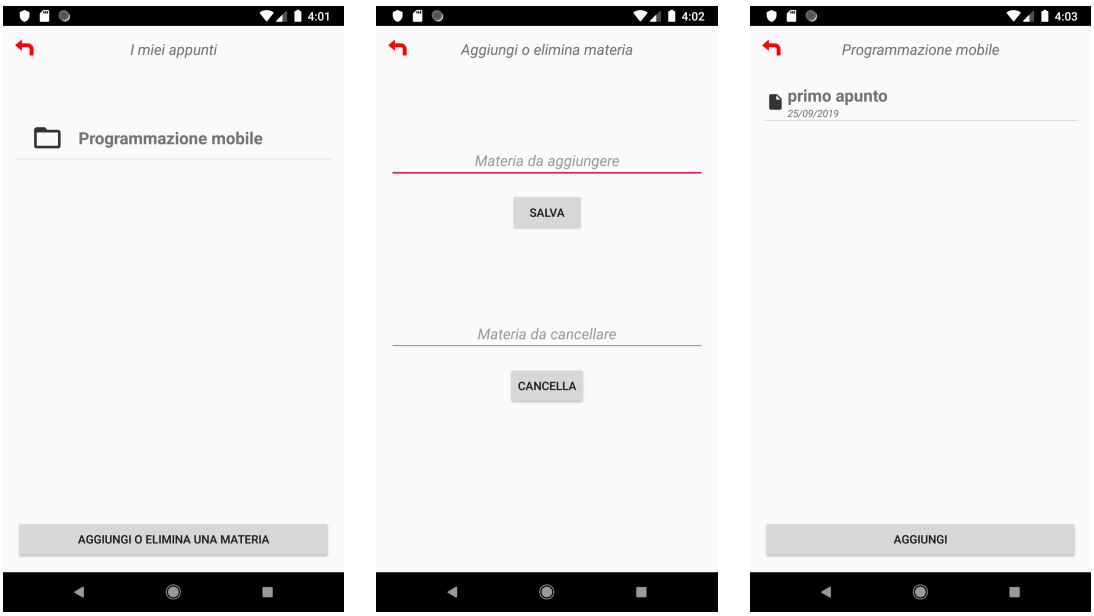

Figura 2.3: Scegli aggiungi o elimina una materia e scegli un appunto.

Scelta la materia accediamo ad una seconda lista contenente tutti gli appunti relativi ad essa; ogni elemento della lsita contiene titolo e data relativi ad ogni appunto. Da questa lista possiamo accedere e visualizzare l'annotazione vera e propria.

Gli appunti possono essere aggiunti con il bottone **"aggiungi"**, qui inseriremo titolo data e annotazione che vogliamo salvare; da questa schermata, inserendo la spunta nel campo **"Condividi"**, possono essere condivisi direttamente gli appunti con gli altri utenti.

Per eliminare un appunto basta tenere premuto l'elemento da eliminare, verrà attivata una finestra di dialogo da cui è possibile confermare la cancellazione.

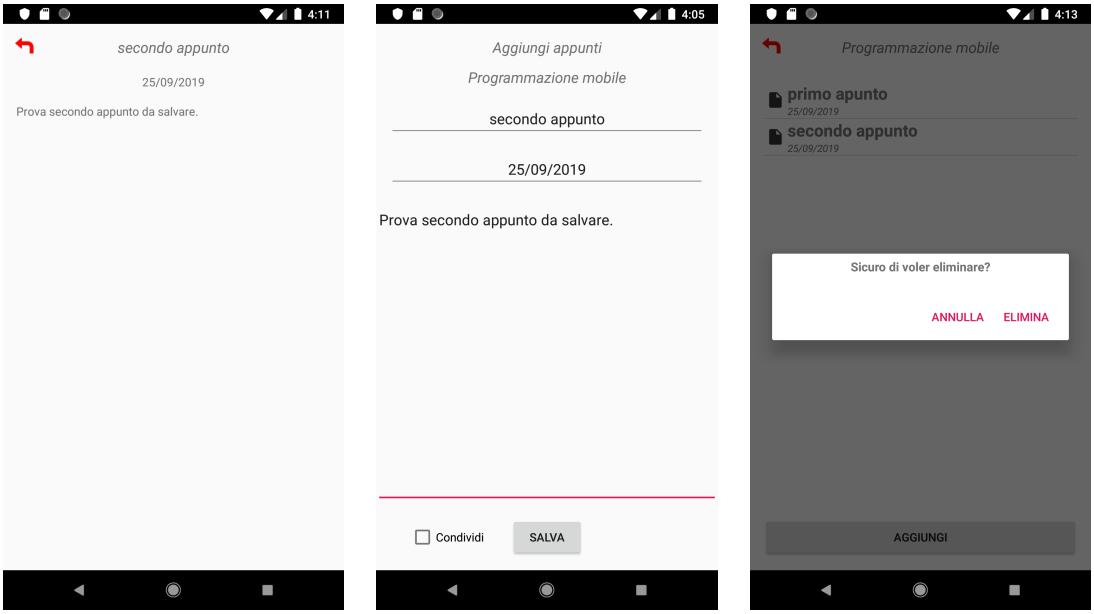

Figura 2.4: Visualizza, aggiungi o elimina un appunto.

2.3 Appunti condivisi

Nella sezione appunti condivisi possiamo scegliere una delle materie predefinite in lista per visualizzare gli appunti che sono stati condivisi ed eventualmente visualizzarli o scaricarli.

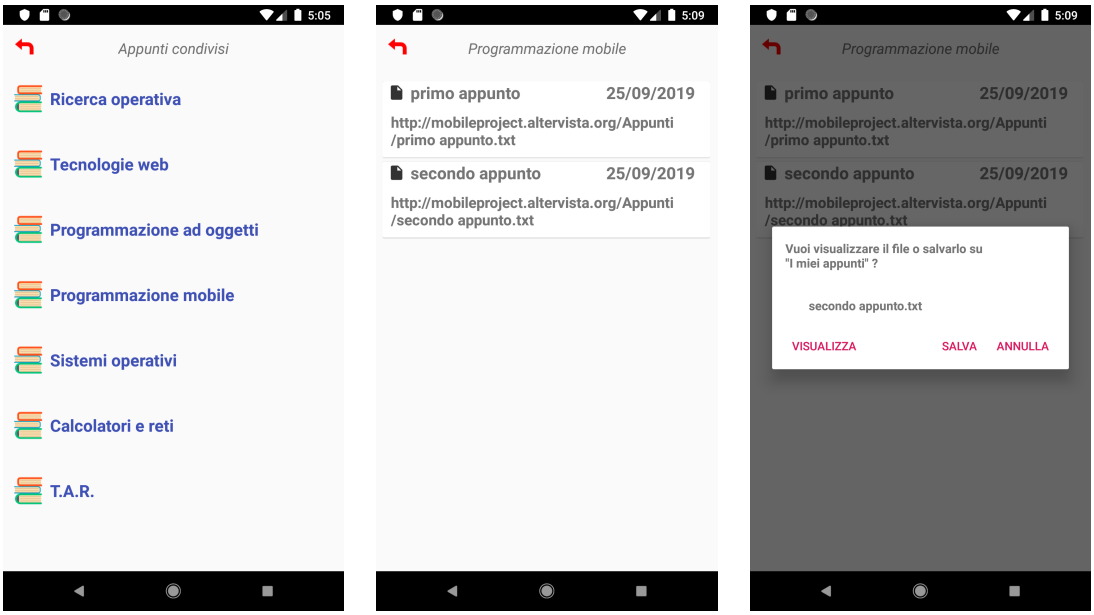

Figura 2.5: Appunti condivisi, scegli la materia, scegli l'appunto, visualizza o scarica.

2.4 Impostazioni

Dalle impostazioni possiamo banalmente modificare **username**, **password** e leggere informazioni **about us**.

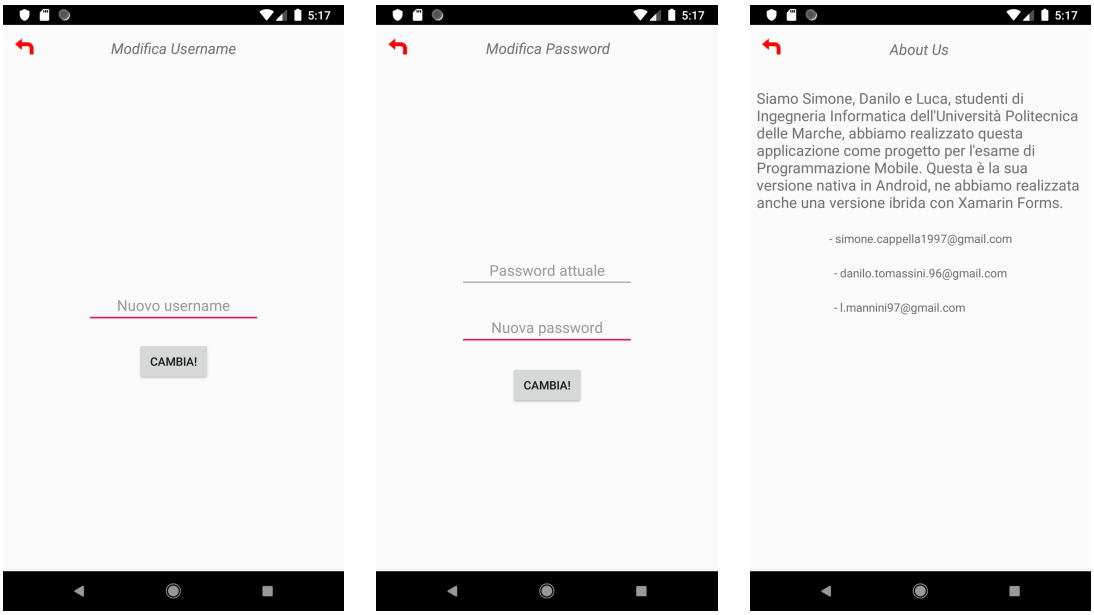

Figura 2.6: Azioni eseguibili nelle impostazioni.

3 Implementazioni Android studio

3.1 Classe SupportTask

Riteniamo opportuno introdurre questa classe e le sue funzionalità ad inizio capitolo poichè essa verrà utilizzata da tutte le funzionalità che richiedono un accesso al database degli utenti.

Notiamo subito che la nostra classe estende la classe astratta **AsyncTask**. Innanzitutto definiamo il significato di task asincrona: una task asincrona è un metodo che non va a lavorare sul thread principale del programma, per cui quest'ultimo può proseguire senza attendere che il metodo chiamato terminati. Nel costruttore inizializziamo il contesto relativo alla classe chiamante. Dal momento che estendiamo una classe astratta dobbiamo sovrascrivere i metodi che ci interessano. Il metodo che andiamo a sovrascrivere noi è *doInBackground* che rappresenta la vera e propria logica del metodo. Non passiamo un numero definito di variabili a tale metodo poichè essi dipenderanno da quale classe del nostro programma effettuerà la chiamata. Vengono utilizzati i tre puntini per indicare che ciò che esso riceverà come parametro sarà un array di stringhe, di nome *params*, di grandezza appunto indefinita. I valori in tale array verrano memorizzati secondo l'ordine in cui vengono inseriti nei parametri al momento della chiamata del metodo. Verrano quindi recuperati accedendo all'indice corrispondente alla posizione del valore nei parametri di chiamata. Il primo parametro che passiamo a questo metodo è sempre *method*, ovvero una stringa sulla quale andremo ad effettuare vari controlli per capire cosa l'utente ha richeisto.

```
\\SupportTask\\
public class SupportTask extends AsyncTask <String, Void, String> {
    Context ctx;

    SupportTask(Context ctx) {
        this.ctx = ctx;
    }

    protected void show(String message) {
        Toast.makeText(ctx, message, Toast.LENGTH_LONG).show();
    }

    @Override
    //Ricevo i parametri in un array di dimensione variabile
    protected String doInBackground(String... params) {
        String line;
        String method = params[0];
[...]
```

Prenderemo in esame soltanto il caso in cui l'utente chieda di registrarsi, gli altri casi sono gestiti in maniera praticamente analoga se non per il numero di parametri che riceviamo e che inviamo al server.

Andiamo ad impostare la connessione, tramite l'oggetto **HttpURLConnection**, e il metodo con il quale invieremo i dati della form. Produciamo poi la stringa che comporrà il body del messaggio al server secondo le regole del metodo post, ovvero "nome_variabile1"=valore1 & "nome_variabile2"=valore2. Vengono inviati i dati e viene chiuso lo stream. Per leggere la risposta del server utilizziamo l'oggetto **InputStream** e il **BufferedReader**, salviamo tutto nella stringa *state* e la ritorniamo alla classe chiamante andando ad invocare il metodo **trim()** che elimina eventuali spazi bianchi all'inizio della stringa.

```
\\SupportTask\\
if (method.equals("register")) {
            String state = "";
            String user_name = params[1];
            String user_pass = params[2];
            String reg_url = params[3];
            try {
                URL url = new URL(reg_url);
                //mi connetto all'URL al quale effettuare la richiesta,
impostando anche il tipo di richiesta
                HttpURLConnection httpURLConnection = (HttpURLConnection)
url.openConnection();
                httpURLConnection.setRequestMethod("POST");
                httpURLConnection.setDoOutput(true);
```

```
                httpURLConnection.setDoInput(true);
                OutputStream OS = httpURLConnection.getOutputStream();
                BufferedWriter bufferedWriter = new
BufferedWriter(new OutputStreamWriter(OS, "UTF-8"));
                //Creo la stringa che compone la richiesta
                String data = URLEncoder.encode("user_name", "UTF-8") + "="
+ URLEncoder.encode(user_name, "UTF-8") + "&" +
                        URLEncoder.encode("user_pass", "UTF-8") + "="
+ URLEncoder.encode(user_pass, "UTF-8");
                bufferedWriter.write(data);
                bufferedWriter.flush();
                bufferedWriter.close();
                OS.close();
                InputStream IS = httpURLConnection.getInputStream();
                BufferedReader bufferedReader = new
BufferedReader(new InputStreamReader(IS, "iso-8859-1"));
                while((line = bufferedReader.readLine()) != null)
                {
                    state += line;
                }
                bufferedReader.close();
                IS.close();
                httpURLConnection.disconnect();
            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            } return state.trim();
[...]
```

3.2 Login e Registrazione

Per avviare il login, ovviamente, è necessario riempire le **EditText** relative a username e password; i relativi valori verranno assegnati alle variabili **login_name** e **login_pass**.

La connessione al database viene instaurata indicando **method**, stringa inizializzata in precedenza a **"login"**, dati relativi ad username e password e l'url contenente il percorso fino alla funzione di login contenuta nel server.

Se la funzione del server da esito negativo non verrà effettuato il login e verrà stampato un **Toast**, per mezzo della funzione **show**. Altrimenti verrà stampato un messaggio di benvenuto e verrà lanciata la **MainActivity**.

```
\\FirstActivity\\
[...]
public void userLogin(View view) {
[...]
try {
  Sring url = "http://mobileproject.altervista.org/login.php";
  auth = supportTask.execute(method, login_name, login_pass, url).get();
  } catch (ExecutionException e)
  {
    e.printStackTrace();
  }catch (InterruptedException a)
  {
    a.printStackTrace();
  }
  if(auth.equals("Login Success"))
  {
    launchMainActivity(view);
    show("Benvenuto "+ login_name + "!");
    finish();
  }else
  {
    show("Dati errati. Riprova.");
  }
}
```

```
[...]
```

Il codice PHP relativo alla funzione di login è mostrato di seguito.

```
<?php
 require 'init.php';
 $user_name = $_POST["login_name"];
 $user_pass = $_POST["login_pass"];
 $sql_query = "SELECT * FROM Utenti WHERE username = '$user_name' AND
password = '$user_pass'";
 $result = mysqli_query($con,$sql_query);
 if(mysqli_num_rows($result)> 0)
 {
 echo "Login Success";
 }
 else
 {
 echo "Login Failed";
 }
 ?>
```

La registrazione viene gestita in modo simile.

In un **if** viene verificato in modo veloce se le condizioni su username e password sono verificate, in caso contrario viene lanciato un avviso. Il metodo con cui si instaura la connessione al server è lo stesso, cambiano solo gli argomenti, **url** e **method**.Una volta che la registrazione è andata a buon fine l'attività viene interrotta e si torna alla schermata di login.

```
\\Register\\
[...]
public void userRegister() {
[...]
if (register_name.length() >= 3 && register_pass.length() >= 3)
{
  try {
    String url = "http://mobileproject.altervista.org/register.php";
    auth= supportTask.execute(method, register_name, register_pass, url).get();
  } catch (ExecutionException e) {
    e.printStackTrace();
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  if (auth.equals("Registrazione avvenuta con successo!")) {
    show("Registrazione effettuata con successo, accedi!");
    finish();
  } else if (auth.equals("Username in uso")) {
    show("Username gia' in uso, prova con uno diverso!");
  }
}
else
{
show(I campi devono contenere almeno 3 caratteri.);
}
[...]
```

Di seguito abbiamo inserito il codice PHP relativo alla funzione di registrazione.

```
<?php
 require "init.php";

 $user_name = $_POST["user_name"];
 $user_pass = $_POST["user_pass"];
 $ver = "SELECT * FROM Utenti WHERE username = '$user_name' ";
 $result = mysqli_query($con,$ver);
 $row = mysqli_fetch_array($result,MYSQLI_NUM);
 if ($row[0] > 0)
  {
    echo "Username in uso";
    }
    else{
```

```
      $sql_query = "INSERT INTO Utenti (username,password)
values('$user_name','$user_pass')";
      if(mysqli_query($con,$sql_query))
        {
            echo"Registrazione avvenuta con successo!";
        }else
            echo"Errore";
          }
 ?>
```

3.3 Orario

Avviata l'activity **orario** vengono catturate le dimensioni del display, calcolata la dimensione del **fragment** contenente la tabella dell'orario e fatto partire uno switch sul giorno corrente.

```
\\Clock\\
[...]
protected void onCreate(Bundle savedInstanceState) {
[...]
Calendar calendar = Calendar.getInstance();
int day = calendar.get(Calendar.DAY_OF_WEEK);
Fragment fragment = null;
switch (day){
  case Calendar.MONDAY:
    fragment = new lun_fragment();
    nav.setSelectedItemId(R.id.lun);
    break;
  case Calendar.TUESDAY:
    fragment = new mar_fragment();
    nav.setSelectedItemId(R.id.mar);
    break;
  case Calendar.WEDNESDAY:
[...]
}
loadFragment(fragment);
[...]
```

Questo switch permette di caricare il fragment relativo al giorno corrente (es. se oggi è lunedì viene caricato il fragment relativo a lunedì); viene, infatti, inizializzato l'oggetto fragment e viene selezionato l'elemento sulla navbar.

```
\\Clock\\
[...]
public boolean onNavigationItemSelected(@NonNull MenuItem item) {
Fragment fragment = null;
switch(item.getItemId())
{
  case R.id.lun:
    next = 0;
    fragment = new lun_fragment();
    break;
  case R.id.mar:
    next = 1;
    fragment = new mar_fragment();
    break;
  case R.id.mer:
    next = 2;
    fragment = new mer_fragment();
    break;
[...]
}
return loadFragment(fragment);
[...]
```

Allo stesso modo la funzione **onNavigationItemSelected**, associa in base allitem della navbar cliccato un'istanza del fragment all'oggetto fragment e nuovamente viene lanciata la funzione **loadFragment**.

La funzione **loadFragment** carica il fragment richiesto e gestisce lo scorrimento a destra o sinistra dipendentemente dalla posizione del fragment corrente rispetto a quello scelto.

```
1  private boolean loadFragment(android.support.v4.app.Fragment fragment)
2      {
3          if(fragment!= null)
4              {
5              FragmentTransaction ft =
6  getSupportFragmentManager().beginTransaction();
7              if(curr < next)
8                  {
9                      ft.setCustomAnimations(R.anim.slide_in,R.anim.slide_out);
10                 }else
11                     {
12                         ft.setCustomAnimations(R.anim.slide_in_left,
13 R.anim.slide_out_right);
14                     }
15             curr = next;
16             ft.replace(R.id.fragment_container, fragment);
17             ft.commit();
18             return true;
19         }
20         return false;
21     }
```

3.3.1 Gestione orario

Prendiamo come riferimento il fragment relativo al **lunedì**, gli altri saranno implementati in modo analogo.

All'interno del fragment vengono nuovamente catturate le dimensioni del display e impostate le dimensioni delle varie caselle della tabella, rispettivamente:

- **Ora:** l'ora di inizio della lezione.
- **Materia:** la materia inserita.
- **Aula:** l'aula nel quale si terrà la lezione.
- **X:** edit per cancellare la riga.

La tabella viene inizializzata nella funzione **onCreate**, qui, ogni elemento della tabella viene assegnato ad una variabile e successivamente "riempito" con il testo salvato, andadolo a recuperare con l'oggetto **sa**, della classe **SalvaOrario**, passando la chiave $\{lun\_1, lun\_2, ..., lun\_11\}$ per quanto riguarda la tabella di lunedì, per la tabella di martedì le chiavi saranno costruite come $\{mar\_1, ..., mar\_11\}$; la stessa logica vale per gli altri giorni; in tal modo possiamo recuperare in modo distinto le 11 materie salvate con la relativa aula per i diversi giorni.

```
1  \\lun_fragment\\
2  [...]
3  public View onCreateView(LayoutInflater inflater,
4  @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
5  [...]
6  SalvaOrario sa = new SalvaOrario();
7  [...]
8  txtMat1 = v.findViewById(R.id.materia1);
9  txtMat1.setText(sa.getMateria("lun_1", getActivity()));
10 txtMat1.setOnClickListener(this);
11 txtAula1 = v.findViewById(R.id.aula1);
12 txtAula1.setText(sa.getAula("lun_1", getActivity()));
13 txtOra1 = v.findViewById(R.id.ora1);
14
15 txtMat2 = v.findViewById(R.id.materia2);
16 txtMat2.setText(sa.getMateria("lun_2", getActivity()));
17 txtMat2.setOnClickListener(this);
18 txtAula2 = v.findViewById(R.id.aula2);
19 txtAula2.setText(sa.getAula("lun_2", getActivity()));
20 txtOra2 = v.findViewById(R.id.ora2);
21 [...]
```

Nella funzione **onClick** viene gestita la scelta di inserire una materia nell'orario o eliminarne una dallo stesso. C'è, infatti, uno switch che gestisce il "click" nelle diverse sezioni della tabella. Abbiamo due diversi tipi di azione:

- **Aggiungere una materia:** per aggiungere una materia si seleziona un campo materia relativo all'ora in cui vogliamo inserirlo, questo attiverà l'elemento corrispondente nello switch che imposterà la variabile *n*, utilizzata successivamente per costruire la chiave e salvare nella classe SalvaOrario, la variabile *ora*, presa direttamente dalla view e chiama la funzione **launchList()**.
- **Elimina una materia:** se viene selezionata la "X" sulla destra della tabella viene semplicemente "pulita" la riga corrispondente, infatti, nello switch, captato l'*edit* corrispondente, viene impostata di nuovo la variabile *n* per essere usata come chiave e le voci *materia* e *aula* vengono messe a *null*, viene lanciata, poi, la funzione **inserisciSalva()**; (la variabile *inc* serve successivamente per inserire in un colpo solo più ore della stessa materia).

```
\\lun_fragment\\
[...]
public void onClick (View v) {
switch (v.getId())
{
  case R.id.materia1:
    n = 1;
    ora = txtOra1.getText().toString();
    v.startAnimation(buttonClick);
    launchList();
    break;
  case R.id.materia2:
    n = 2;
    ora = txtOra2.getText().toString();
    v.startAnimation(buttonClick);
    launchList();
    break;
[...]
  case R.id.edit1:
    n = 1;
    v.startAnimation(buttonClick);
    materia = null;
    aula = null;
    inc = 1;
    inserisciSalva();
    break;
  case R.id.edit2:
    n = 2;
    v.startAnimation(buttonClick);
    materia = null;
    aula = null;
    inc = 1;
    inserisciSalva();
    break;
[...]
```

La finzione **launchList**, chiamata nel momento in cui si vuole aggiungere una materia all'orario non fa altro che lanciare un **intent esplicito**, questo ci permette di scambiare dati tra l'activity chiamante e la chiamata, infatti tra i parametri della funzione **putExtra** abbiamo una chiave, *giorno_ora* e un valore, *"lunedi'" + ora* (*ora* è stata assegnata in precedenza nello switch). Inoltre c'è bisogno di un *REQUEST_CODE* utilizzato come chiave di riconoscimento tra le activity.

```
\\lun_fragment\\
[...]
public static final int REQUEST_CODE = 0000;
[...]
public void launchList() {
  Intent intent = new Intent (getActivity(), List.class);
  intent.putExtra("giorno_ora", "lunedi'  " + ora);
  startActivityForResult(intent, REQUEST_CODE);
}
```

L'activity **List** genera la lsita predefinita delle materie da cui si può scegliere quella da inserire nella casella dell'orario scelta.

Qui i dati vengono recuperati proprio grazie alla chiave *giorno_ora*, successivamente vengono istaziati *myDialog* e *adapter*.

La funzione **setOnItemClickListener** chiamata per mezzo dell'oggetto *listMaterie* e la successiva **OnItemClick** permettono di captare quale materia della lista viene scelta grazie alla sua posizione nella lista stessa.

Scelta la materia viene aperta la finestra di dialogo, all'interno di questa vengono definiti diversi elementi come una editText per l'aula e due bottoni, −+, questi permettono di incrementare e decrementare le ore di lezione relative alla materia.

Una volta inserita l'aula e il numero di ore si provede al salvataggio. Un listener sul bottone *btnSalva* fa si che quando venga restituito il risultato all'activity chiamante con la variabile *intent*, questa contiene la materia scelta, l'aula e la conta delle ore, l'attivity termina con finish(). Una cosa

analoga avviene nel momento in cui si decide tramite l'apposito bottone di inserire una nuova materia non contenuta nella lista predefinita.

```
\\List\\
[...]
Intent intent = getIntent();
String giorno_ora = intent.getStringExtra("giorno_ora");
[...]
myDialog = new Dialog(this);
final ArrayAdapter<String>  adapter = new ArrayAdapter<>(this,
android.R.layout.simple_list_item_1, android.R.id.text1, listItem);
listMaterie.setAdapter(adapter);
[...]
listMaterie.setOnItemClickListener(new AdapterView.OnItemClickListener() {
  @Override
  public void onItemClick(AdapterView<?> parent, View view, int position,
  long id) {
    value = adapter.getItem(position);
[...]
myDialog.show();
[...]
btnMeno.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
    i--;
    textContatore.setText(Integer.toString(i));
  }
}
[...]
btnPiu.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
  i++;
  textContatore.setText(Integer.toString(i));
  }
}
[...]
btnSalva.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
[...]
    Intent intent = new Intent();
    intent = intent.putExtra("mat", valore);
    setResult(Activity.RESULT_OK, intent);
    finish();
    myDialog.dismiss();
  }
[...]
```

La funzione **onActivityResult** si occupa di reperire dall'activity Lsit i risultati ottenuti dalla scelta. L'array di stringhe *res* contiene ora nella posizione 0 la materia, nella posizione 1 l'aula e nella posizione 2 il numero di ore, che va "parsato" in un intero in quanto arrivava come una Stringa.

Viene considerato il caso in cui non si completa la scelta, in tal caso nell'activity Lista viene impostato il nome della materia a *back* e non viene eseguita nessun'altra azione, la routine si interrompe. Nel caso in cui la scelta è stata completata i dati raccolti andranno salvati, sono passati, dunque, alla funzione **salvaOrario**.

```
\\lun_fragment\\
[...]
public void onActivityResult(int requestCode, int resultCode, Intent data){
  super.onActivityResult(requestCode, resultCode, data);
  if ((requestCode == REQUEST_CODE) && (resultCode == Activity.RESULT_OK)) {
    String[] res = data.getStringArrayExtra("mat");
    materia = res[0];
    aula = res[1];
    inc = Integer.parseInt(res[2]);
```

```
10    }
11    if (materia.equals("back")){}
12    else{
13      salvaOrario(i, materia, aula);
14    }
15  }
16  [...]
```

La funzione **salvaOrario** va a definire un cursore che sarà popolato con l'elemento del database trovato dal metodo **searchM** (questo trova nel database l'elemento il cui campo materia ha lo stesso nome della materia inserita). Questo cursore permette di verificare, grazie all'*if* se la materia inserita è gia presente nel database contenente le materie, in caso positivo il vecchio elemento viene eliminato; successivamente si inserisce nel database delle materie il nuovo elemento e viene chiamata la funzione **inserisciSalva()**.

Le materie vengono inserite nel database delle materie per permettere alla sezione *mieiappunti* di rendere disponibili le materie inserite nell'orario.

```
1  \\lun_fragment\\
2  [...]
3  public void salvaOrario (String key, String materia, String aula) {
4    Cursor c;
5    c = dm.searchM(materia);
6    if (c.getCount() > 0) {
7      dm.delete(materia);
8    }
9    dm.insert(materia, ora, aula, key);
10   inserisciSalva();
11 [...]
```

La funzione **inserisciSalva** permette di salvare, nuovamente tramite i metodi della classe **Salva Orario**, materia e aula scelti e nuovamente di aggiornare i valori delle *txtMat* e *txtAula*.

Il **while** all'interno della funzione ripete il ciclo finchè la variabile *inc* che conteneva la conta delle ore non arriva a 0; la Stringa *q* viene costruita in modo da diventare una chiave per i metodi della classe SalvaOrario, infatti la variabile *i* contiele la stringa "lun_" e la variabile *n* contiene il numero dipendentemente dalla riga scelta dalla tabella. Le variabili *n* ed *inc* alla fine del ciclo vengono rispettivamente incrementata e decrementata, questo permette di salvare in un solo colpo stessa materia e stessa aula all'interno della tabella, infatti il numero dell'elemento della tabella incrementa finche non si esaurisce il numero di ore deciso nella finestra di dialogo nell'activity List.

Lo **switch** su *n* ha il compito di aggiornare immediatamente il valore della tabella nella posizione *n*.

```
1  \\lun_fragment\\
2  [...]
3  public void inserisciSalva(){
4    while (inc > 0){
5      String q = i + n;
6      sa.setMateria(q, materia, getActivity());
7      sa.setAula(q, aula, getActivity());
8      switch (n){
9        case 1:
10         txtMat1.setText(sa.getMateria(q, getActivity()));
11         txtAula1.setText(sa.getAula(q, getActivity()));
12         break;
13       case 2:
14         txtMat2.setText(sa.getMateria(q, getActivity()));
15         txtAula2.setText(sa.getAula(q, getActivity()));
16         break;
17 [...]
18   }
19   n++;
20   inc--;
21 }
22 [...]
```

La classe **SalvaOrario** gestisce salvataggio ed estrazione per mezzo delle **SharedPreferences** di materia e aula.

Al suo interno sono presenti, infatti, quattro metodi:

- **setMateria:** prende tra gli argomenti chiave e materia e va a salvare la materia con tale chiave.
- **getMateria:** va ad estrarre dai salvataggi la materia corrispondente alla chiave di ricerca.
- **setAula:** di nuovo prende tra gli argomenti chiave ed aula (in questo caso la chiave viene combinata con la stringa *"_A"* per caratterizzare le chiavi relative alle aule) salva, dunque, il nome dell'aula.
- **getAula:** estrae l'aula dai salvataggi per mezzo della chiave combinata con la stringa di cui sopra.

```
1  \\SalvaOrario\\
2  [...]
3  public static void setMateria(String key, String value,  Context context) {
4    SharedPreferences.Editor editor = preferences.edit();
5    editor.putString(key, value);
6    editor.commit();
7  }
8  public static String getMateria(String key, Context context)
9  {
10   return preferences.getString(key, null);
11 }
12 public static void setAula(String key, String value,  Context context) {
13   SharedPreferences.Editor editor = preferences.edit();
14   editor.putString(key+"_A", value);
15   editor.commit();
16 }
17 public static String getAula(String key, Context context) {
18   return preferences.getString(key+ "_A", null);
19 }
```

3.4 I miei appunti

Avviata l'activity **i miei appunti** viene mostrata la lista delle materie salvate nell'applicazione, queste sono il risultato delle materie aggiunte nell'orario o eventualmente aggiunte nella sezione in questione.

All'avvio si definisce e istanzia un cursore che va ad estrarre tutti gli elementi dal database e con un *CursorAdapter* va a popolare la lsita. Nuovamente viene utilizzato un *onItemClickListener* che permette di catturare l'elemento dalla lista, viene, dunque, chiamata l'activity **Notes_Page** con un intent passando il parametro *a*, stringa che contiene il nome della materia scelta.

```
\\MyNotes\\
[...]
protected void onCreate(Bundle savedInstanceState) {
[...]
cursor = dm.selectAll();
[...]
adapter = new SimpleCursorAdapter(this, R.layout.list_mat, cursor,
  fromColumns, viewsList, 0);
listMat.setAdapter(adapter);
listMat.setOnItemClickListener(new AdapterView.OnItemClickListener() {
  @Override
  public void onItemClick(AdapterView<?> parent, View view,
    int position, long id) {
    Cursor c = adapter.getCursor();
    String a = c.getString(2);
    Intent i = new Intent(getApplication(), Notes_Page.class);
    i.putExtra("mat", a);
    startActivity(i);
  }
}
[...]
```

3.4.1 Aggiungi una materia

Possiamo aggiungere manualmente una nuova materia o, eventualmente, eliminarne una contenuta nel database attivando il relativo bottone. Viene lanciata con un intent esplicito l'activity **AggMaterie**; si utilizza un intent esplicito per permettere l'aggiornamento della lista una volta aggiunta o eliminata una materia dalla stessa. Nella funzione **onActivityResult**, infatti, appurata la validità del ritorno dall'activity si estrae il dato, *res* contiene il nome della materia aggiunta, e definito un nuovo cursore *nc* si applicano metodi quali **changeCursor** sull'adapter in modo da indicargli il nuovo cursore da utilizzare e la **notifyDataSetChanged**; in tal modo la lista verrà aggiornata.

```
\\MyNotes\\
[...]
public static final int REQUEST_CODE2 = 2222;
[...]
private void launchAggMat() {
  Intent intent = new Intent(this, AggMaterie.class);
  intent.putExtra("app", "agg materie");
  startActivityForResult(intent, REQUEST_CODE2);
}
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data){
  super.onActivityResult(requestCode, resultCode, data);
  if ((requestCode == REQUEST_CODE2) && (resultCode == Activity.RESULT_OK)) {
    String res = data.getStringExtra("res");
    Toast.makeText(getApplicationContext(), res, Toast.LENGTH_SHORT).show();
  }
  Cursor nc;
  nc = dm.selectAll();
  adapter.changeCursor(nc);
  adapter.notifyDataSetChanged();
}
[...]
```

L'activity **AggMaterie** consiste di una *editText* e due bottoni, uno per salvare la materia inserita nella edit e una per eliminarla. Nella funzione **onCreate** dell'activity AggMaterie viene "raccolto" l'intent.

Le funzioni relative al salvataggio ed eliminazione sono implementate direttamente nella funzione **onClick**, con uno switch sull'elemento della vista cliccato.

- **Salva:** viene estratta la stringa e istanziato un cursore; tramite il cursore si va a cercare nel database un elemento il cui campo materia sia uguale alla materia inserita, nell'**if** viene dunque controllato tramite il metodo **getCount** sul cursore se tale elemento è stato trovato o meno, in caso positivo viene solo lanciato un avviso, altrimenti può essere salvata la nuova materia e terminata l'activity impostando il risultato dell'intent.
- **Elimina:** si estrae nuovamente la stringa dalla edit, si istanzia il cursore e nuovamente si procede alla ricerca di un elemento il cui campo materia corrisponda alla materia scelta, stavolta nel caso in cui venga trovata viene eliminata e terminata l'activity.

```
\\AggMaterie\\
[...]
protected void onCreate(Bundle savedInstanceState) {
[...]
  Intent intent = getIntent();
}
[...]
@Override
public void onClick(View v) {
  switch (v.getId()){
[...]
    case R.id.btnInsert:
      materia = editMateria.getText().toString();
      Cursor c = dm.searchM(materia);
      if(c.getCount() > 0){
        [...]
      }else{
        dm.insert(materia, "ora", "aula", "code");
        Intent res  = new Intent();
        res = res.putExtra("res", materia);
        setResult(Activity.RESULT_OK, res);
        finish();
      }
      break;
    case R.id.btnDelete:
      String materia = editMateria.getText().toString();
      Cursor del = dm.searchM(materia);
      if (del.getCount() > 0){
        dm.delete(materia);
        finish();
      }
      [...]
      break;
  }
}
[...]
```

3.4.2 Vedi appunti

Scelta la materia dalla lista visualizzata all'avvio di **MyNotes** viene lanciata, come visto, l'activity **Notes_Page**.

All'avvio viene subito estratto il dato passato dall'activity chiamante, si tratta del nome della materia scelta, successivamente vengono definiti un **cursore** che contiene gli elementi del database la cui materia coincide con quella ricevuta in *a* e un **adapter** che permette di riempire la lista con gli elementi del cursore, in particolare vengono messi nella lista titolo e data dell'appunto.

```
\\Notes_Page\\
[...]
protected void onCreate(Bundle savedInstanceState) {
```

```
  [...]
  Intent r = getIntent();
  a = r.getStringExtra("mat");
  [...]
  cursor = da.searchM(a);
  adapter = new SimpleCursorAdapter(this, R.layout.list_app_loc, cursor,
    fromColumns, viewsList, 0);
  listNote.setAdapter(adapter);
  [...]
```

In tal caso abbiamo due possibili eventi sulla lista, un click singolo o un click prolungato (longClick).

- **Click:** la funzione **onItemClick** gestisce il click singolo sull'elemento, grazie alla posizione viene istanziato un nuovo cursore all'elemento in questione e ne vengono estratti come stringhe *materia*, *titolo*, *datael'appuntoveroeproprio*, viene, dunque, stampato a schermo un **Toast** contenente il titolo dell'appunto scelto e viene lanciata la funzione **launchVedi**.
- **LongClick:** il long click permette di eliminare l'appunto. Viene nuovamente istanziato un cursore attraverso il quale vado ad estrarre *code* e *titolo* dell'appunto selezionato.

  Viene successivamente costruita una finestra di dialogo grazie alla classe AlertDialog. Nell'alert dialog devono essere settati i bottoni "positivo" e "negativo", rispettivamente grazie a **setPositiveButton** e **setNegativeButton**.

  Settato il positive button al suo interno viene utilizzata una funzione **onClick** che al click fa partire il metodo **delete** della classe **DataAppLoc**, database per gli appunti, che va a cancellare l'appunto andandolo a cercare con la variabile *code*. Successivamente viene aggiornata la lista come nelle activity precedenti e costruita la stringa con il percorso del file questo viene cancellato.

  Il set del negative button non implementa nessuna funzione, in tal caso la alert dialog viene chiusa.

```
\\Notes_Page\\
[...]
da = new DataAppLoc(this);
[...]
listNote.setOnItemClickListener(new AdapterView.OnItemClickListener() {
  @Override
  public void onItemClick(AdapterView<?> parent, View view, int position,
    long id) {
    Cursor c = adapter.getCursor();
    code = c.getString(0);
    String materia = c.getString(1);
    String titolo = c.getString(3);
    String data = c.getString(2);
    String app = c.getString(4);
    Toast.makeText(getApplicationContext(), code, Toast.LENGTH_SHORT).show();
    launchVedi(materia, titolo, data, app);
  }
  @Override
  public boolean onItemLongClick(AdapterView<?> parent, final View view,
    final int position, long id) {
    Cursor c = adapter.getCursor();
    code = c.getString(0);
    final String titolo = c.getString(3);
    [...]
    LayoutInflater inflater = LayoutInflater.from(context);
    View mess = inflater.inflate(R.layout.messaggio_elimina,null);
    AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(context);
    alertDialogBuilder.setView(mess);
    alertDialogBuilder.setPositiveButton("ELIMINA",
      new DialogInterface.OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
        da.delete (code);
        Cursor nc;
```

```
        nc = da.searchM(a);
        adapter.changeCursor(nc);
        adapter.notifyDataSetChanged();
        Context context = getApplicationContext();
        String folder = context.getFilesDir().getAbsolutePath()
          + File.separator + "Appunti/";
        File file = new File(folder + titolo + ".txt");
        boolean deleted = file.delete();
      }
    }
    alertDialogBuilder.setNegativeButton("ANNULLA",
      new DialogInterface.OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {}
    }
    [...]
```

Come già detto nel caso di un singolo click su un elemento della lista viene chiamata la funzione **launchVedi**, questa crea un intent e un bundle contenente nome della materia cui si riferisce, titolo dell'appunto, data e annotazioni, questo bundle viene inserito nell'intent con il quale sarà chiamata l'activity **VediAppunti**.

```
\\Notes_Page\\
[...]
public void launchVedi(String materia, String titolo, String data, String app){
  Intent i = new Intent(getApplication(), VediAppunti.class);
  Bundle bundle = new Bundle();
  bundle.putString("materia", materia);
  bundle.putString("titolo", titolo);
  bundle.putString("data", data);
  bundle.putString("app", app);
  i.putExtra("data", bundle);
  startActivity(i);
}
```

Nel caso in cui si voglia aggiungere un appunto va attivato il bottone **Aggiungi**, il cui evento è catturato nella funzione **onClick** della classe; questa banalmente chiama la funzione **launchAggiungi**.

La funzione in questione si occupa semplicemente di utilizzare un intent esplicito che va a lanciare l'activity **AggiungiAppuntiLoc**

```
\\Notes_Page\\
[...]
public void launchAggiungi(){
  Intent intent = new Intent(this, AggiungiAppuntiLoc.class);
  intent.putExtra("app", a);
  startActivityForResult(intent, REQUEST_CODE1);
}
[...]
```

L'activity **AggiungiAppuntiLoc** si occupa di salvare gli appunti in locale ed eventualmente nella sezione per gli appunti condivisi.

Per quanto riguarda il salvataggio in locale, nella funzione **onClick** della classe nel caso in cui venga attivato il bottone *salva* vengono salvati i testi inseriti delle edit e i dati vengono salvati nel database relativo agli appunti, viene, quindi, restituito l'intent con la variabile *res* con una semplice stringa "*Appuntisalvati*".

```
\\AggiungiAppuntiLoc\\
[...]
public void onClick(View v) {
[...]
  case R.id.btnSalvaApp:
    final String titolo = editTitolo.getText().toString();
    String data = editData.getText().toString();
    String appunti = editApp.getText().toString();
    da.insert(a, data, titolo, appunti);
    String res = "Appunti salvati";
    Intent intent = new Intent();
```

```
12      intent = intent.putExtra("res", res);
13      setResult(Activity.RESULT_OK, intent);
14  [...]
```

Terminata l'activity per salvare nuovi appunti si torna alla Notes_Page, qui la funzione **onActivityResult** come al solito controlla il risultato dell'activity chiamata con il *REQUEST_CODE* e il *RESULT_OK*, verificata la validità viene stampato il messaggio per la conferma del salvataggio e successivamente aggiornata la view della lista.

```
1  public void onActivityResult(int requestCode, int resultCode, Intent data){
2    [...]
3    if ((requestCode == REQUEST_CODE1) && (resultCode == Activity.RESULT_OK)) {
4      String res = data.getStringExtra("res");
5      Toast.makeText(getApplicationContext(), res, Toast.LENGTH_SHORT).show();
6    }
7    Cursor nc;
8    nc = da.searchM(a);
9    adapter.changeCursor(nc);
10   adapter.notifyDataSetChanged();
11 }
12 [...]
```

3.5 Appunti condivisi

3.6 Impostazioni

La sezione per le impostazioni è molto basilare. Le voci che vanno a comporre il suo menù sono tre: Modifica Username, Modifica Password e About Us. Il tutto viene gestito con una ListView. Quando viene effettuato un click su una delle tre voci viene lanciata l'activity corrispondente tramite la funzione **setOnItemClickListener** che associa all'oggetto *lv*, che è una **ListView**, un listener. Un listner è un gestore degli eventi, che cattura, in questo caso, il click su un item della nostra ListView. Come parametro è necessario passargli un oggetto **Listner** che andiamo a istanziare direttamente inline. Successivamente è necessario sovrascrivere il metodo **OnItemClick** fornitoci dall'interfaccia **AdapterView.OnItemClickListener** inserendo al suo interno la logica del nostro gestore eventi.

Il metodo **OnItemClick** ci viene in aiuto estrapolando dal click dell'utente il numero sequenziale dell'elemento della ListView cliccato ed inserendolo nella variabile intera *position*, ad esempio: se l'utente cliccherà sulla voce **Modifica Username**, che è la prima dall'alto, il valore di *position* sarà pari a 0. Lo **switch** gestisce i diversi casi chiamando la funzione apposita per l'apertura dell'activity desiderata.

```
\\Settings\\
lv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
  @Override
  public void onItemClick(AdapterView<?> parent, View view, int position,
  long id)
  {
          switch(position)
                {
                  case 0:
                          launchEditUser(view);
                          break;
                  case 1:
                          launchEditPass(view);
                          break;
                        case 2:
                          launchAboutUs(view);
                          break;
                }
  }
```

3.6.1 Modifica Username

Activity minimal per quando riguarda la modifica dello username. Essa presenta una **EditText** nel quale inserire il nuovo username desiderato e un **Button** per eseguire l'operazione di modifica. Al click sul bottone parte una procedura che innanzitutto verifica se lo username inserito rispetta le condizioni sulla lunghezza del campo e che fa poi partire il metodo **execute** della classe **SupportTask** per effettuare la modifica vera e propria sui nostri database. Il codice php è strutturato in modo che ci ritorni delle stringhe che andremo a prendere e inserire nella variabile *auth*. Sulla base del valore di *auth* verrano mostrati avvisi all'utente che andranno ad indicare se l'operazione è andata a buon fine o meno. Nel blocco **catch** vengono gestiti i casi in cui vengano lanciate delle eccezioni dal codice presente nel blocco **try**.

```
\\EditUser\\
[...]
String method = "edit";
                String newusr = txtNewusr.getText().toString();
                String auth = "";
                SupportTask supportTask = new SupportTask(EditUser.this);
                if (newusr.length() >= 3)
                {
                    try
                    {
                        String url = "http://mobileproject.altervista.org/
editusername.php";
                        auth = supportTask.execute(method, FirstActivity.
login_name, newusr, url).get();
```

```
      FirstActivity.login_name = newusr;
                    }catch (ExecutionException e)
                    {
                        e.printStackTrace();
                    }catch (InterruptedException a)
                    {
                        a.printStackTrace();
                    }
                    switch(auth) {
                        case "modificato":
                            Toast.makeText(this, "Username cambiato con
successo!", Toast.LENGTH_SHORT).show();
                            Toast.makeText(this, "Per cambiare nome nella home
riavviare l'applicazione.", Toast.LENGTH_SHORT).show();
                            break;
                        case "in uso":
                            Toast.makeText(this, "Username in uso!", Toast.
LENGTH_SHORT).show();
                            break;
                        default:
                            Toast.makeText(this, "Errore, la preghiamo di
riprovare in seguito.", Toast.LENGTH_SHORT).show();
                            break;
                    }
                }else
                {
                    Toast.makeText(this, "L'username deve essere minimo di 3
caratteri.",Toast.LENGTH_SHORT).show();
                }
[...]
```

Il codice PHP è molto semplice, riceve tramite metodo POST due variabili, lo username attuale, e il nuovo username desiderato dall'utente. Effettua innanzitutto un controllo per verificare che le variabili ricevute non siano nulle, poi controlla che il nuovo username sia disponibile, in caso affermativo effettua la query di update al database stampando un risultato di modifica avvenuta.

```
\\editusername.php\\
<?php
require 'init.php';
$oldusername = $_POST['oldusername'];
$newusername = $_POST['newusername'];
$querycheck = "SELECT * FROM Utenti WHERE username = '$newusername'";
$querycheckold = "SELECT * FROM Utenti WHERE username = '$oldusername'";
$queryupdate = "UPDATE Utenti SET username = '$newusername' WHERE username = '$oldusername' ";
$result = mysqli_query($con, $querycheck);
$resultold = mysqli_query($con, $querycheckold);
$num = mysqli_num_rows($resultold);
//echo $num;
if($oldusername!= null && $newusername!= null)
{
  if(mysqli_num_rows($result) > 0)
    {
      echo "in uso";
    }else if ($num == 1)
      {
          mysqli_query($con, $queryupdate);
          echo "modificato";
        } else
          {
              echo 'username attuale errato';
            }
}else
  {
      echo "campi errati";
    }
?>
```

3.6.2 Modifica Password

Il funzionamento di tale sezione dell'applicazione è del tutto simile a quello per la modifica dello username. Anche qui, come in Modifica Username, è stato necessario recuperare l'username dell'utente attualmente loggato, per poter effettuare le query sul giusto record del database. Qui il nome utente viene recuperato direttamente dalla funzione **execute** accedendo alla variabile statica *login_name* che viene valorizzata non appena l'utente effettua il login. Una variabile statica è una variabile accessibile da ogni punto dell'applicazione per tanto potente quanto pericolosa. Riportiamo di seguito il codice PHP che si occupa della modifica della password.

```
1  <?php
2  require 'init.php';
3  $username = $_POST['username'];
4  $oldpass = $_POST['oldpass'];
5  $newpass = $_POST['newpass'];
6  $querycheck = "SELECT * FROM Utenti WHERE username = '$username' AND
7  password = '$oldpass'";
8  $queryupdate = "UPDATE Utenti SET password = '$newpass' WHERE
9  username = '$username'";
10 $resultcheck = mysqli_query($con, $querycheck);
11 if ($oldpass!= null & $newpass != null)
12 {
13   if(mysqli_num_rows($resultcheck) > 0)
14     {
15       mysqli_query($con, $queryupdate);
16         echo "updated";
17     }else
18     {
19       echo 'pass errata';
20     }
21 }else
22 {
23   echo 'errore';
24 }
25 ?>
```

3.7 Database

All'interno dell'applicazione vengono utilizzati due database locali, **DataAppLoc** e **DataManager**.

DataManager viene utilizzata per salvare le materie; in particolare l'orario utilizza questo database per salvare le materie inserite nell'orario stesso, senza ripetizioni, e la sezione miei appunti, per popolare la lista delle materie o eventualmente per aggiungere una materia in modo da selezionarla e accedere agli appunti relativi alla materia stessa.

DataAppLoc viene utilizzata per salvare gli appunti; questo database viene utilizzato dalla sezione miei appunti, qui, scelta la materia si va a salvare un appunto indicando titolo e data, questo viene, dunque, salvato nel database in questione.

Entrambi estendono la classe **SQLiteOpenHelper**, questa permette di utilizzare funzioni sqlite implementando le funzioni **SQLiteOpenHelper**, **onCreate**, **onUpdate**.

3.7.1 DataAppLoc

La classe DataManager definisce alcune variabili **static final**

- **TABLE_ROW_ID:** che contiene l'ID univoco dell'elemento in tabella.
- **TABLE_ROW_M:** stringa che contiene il nome della materia di cui si sta salvando un appunto.
- **TABLE_ROW_D:** stringa che contiene la data.
- **TABLE_ROW_T:** stringa che contiene il titolo.
- **TABLE_ROW_A:** stringa che contiene il testo dell'appunto vero e proprio.
- **DB_NAME:** nome del database, utilizzato per fare riferimento agli elementi della superclasse.

- **DB_VERSION:** versione del database, inizializzata a 1, variabile utilizzata, fondamentalmente per fare riferimento agli elementi della superclasse.
- **TABLE_M_D_A:** vero e proprio nome del database utilizzato per costruire le stringhe con le funzioni sql.

```
\\DataAppLoc\\
[...]
public class DataAppLoc extends SQLiteOpenHelper {
  public static final String TABLE_ROW_ID = "_id";
  public static final String TABLE_ROW_M = "m";
  public static final String TABLE_ROW_D = "d";
  public static final String TABLE_ROW_T = "t";
  public static final String TABLE_ROW_A = "a";
  private static final String DB_NAME = "m_d_t_a_db";
  private static final int DB_VERSION = 1;
  private static final String TABLE_M_D_A = "m_and_d_and_a";
[...]
@Override
public void onCreate(SQLiteDatabase db) {}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {}
[...]
```

La funzione **DataAppLoc** è il costruttore della classe e la sottoclasse **CustomSQLiteOpenHelper** è un helper, una classe astratta, utilizzata per gestire, creare e versionare il database; per essere utilizzata questa sottoclasse ha bisogno dell'implementazione dei metodi **onCreate(SQLiteDatabase)** e **onUpgrade(SQLiteDatabase)**, la onCreate, si occupa, appunto, di creare, nel caso non esista già, il database; la stringa *query* viene utilizzata per generare la stringa con i relativi comandi SQL. Allo stesso modo sono necessarie per la classe DataAppLoc le versioni di onCreate e onUpgrade nell'estratto di codice precedente.

```
\\DataAppLoc\\
[...]
private SQLiteDatabase db;
[...]
public DataAppLoc(Context context){
  super(context, DB_NAME, null, DB_VERSION);
  CustomSQLiteOpenHelper helper = new CustomSQLiteOpenHelper (context);
  db = helper.getWritableDatabase();
}
[...]
private class CustomSQLiteOpenHelper extends SQLiteOpenHelper{
  public CustomSQLiteOpenHelper(Context context){
  super(context, DB_NAME, null, DB_VERSION);
}

@Override
public void onCreate(SQLiteDatabase db){
  String query = "create table " + TABLE_M_D_A + " (" +
    TABLE_ROW_ID + " integer primary key autoincrement not null, " +
    TABLE_ROW_M + " text not null, " +
    TABLE_ROW_D + " text not null, " +
    TABLE_ROW_T + " text not null, " +
    TABLE_ROW_A + " text not null);";
            db.execSQL(query);
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){}
[...]
```

Per quanto riguarda le funzioni di inserimento ricerca e cancellazione la logica è sempre la stessa, si tratta di costruire una stringa in modo da avere al suo interno una funzione SQL da utilizzare nel metodo **execSQL**.

- **insert:** inserisce un nuovo elemento nel database andando a riempire ogni tabella con i parametri passatigli nella chiamata.

- **delete:** va ad eliminare l'elemento dal database il cui id corrisponde al parametro *i* passato al metodo nella chiamata dello stesso.
- **searchM:** cerca nel database l'elemento la cui tabella della materia corrisponde alla stringa *m*, nome della materia passata come parametro, questo metodo ritorna il cursore che "punta" all'elemento trovato.
- **searchTitlo:** cerca l'elemento il cui titolo corrisponde alla stringa passata come argomento, anch'esso ritorna il cursore.

```
\\DataAppLoc\\
[...]
public void insert(String m, String d, String t, String a){
  String query = "INSERT INTO " + TABLE_M_D_A + " (" +
    TABLE_ROW_M + ", " + TABLE_ROW_D + ", " +
    TABLE_ROW_T + ", " + TABLE_ROW_A + ") " + "VALUES
    (" + "'" + m + "'" + ", " + "'" + d + "'" + ", " + "'" + t + "'" + ", " +
     "'" + a + "'" + ");";
  db.execSQL(query);
}
public void delete(String i){
  String query = "DELETE FROM " + TABLE_M_D_A +
    " WHERE " + TABLE_ROW_ID + " = '" + i + "';";
  db.execSQL(query);
}
public Cursor searchM(String m){
  String query = "SELECT " + TABLE_ROW_ID + ", " +
    TABLE_ROW_M + ", " + TABLE_ROW_D + ", " +
    TABLE_ROW_T + ", " + TABLE_ROW_A + " from " +
    TABLE_M_D_A + " WHERE " + TABLE_ROW_M + " = '" + m + "';";
  Cursor c = db.rawQuery(query, null);
  return c;
}
public Cursor searchTitolo(String t){
  String query = "SELECT " + TABLE_ROW_ID + ", " +
    TABLE_ROW_M + ", " + TABLE_ROW_D + ", " +
    TABLE_ROW_T + ", " + TABLE_ROW_A + " from " +
    TABLE_M_D_A + " WHERE " + TABLE_ROW_T + " = '" + t + "';";
  Cursor c = db.rawQuery(query, null);
  return c;
}
```

3.7.2 DataManager

DataManager ha la stessa struttura di **DataAppLoc**. Le tabelle di questo database sono:

- **TABLE_ROW_ID:** contiene l'ID univoco dell'elemento in tabella.
- **TABLE_ROW_C:** contiene un codice, in precedenza utilizzato per assegnare il salvataggio ad una casella della tabella orario.
- **TABLE_ROW_M:** nome della materia.
- **TABLE_ROW_O:** orario di inizio della lezione.
- **TABLE_ROW_A:** aula in cui si tiene la lezione.
- **DB_NAME:** nome del database, utilizzato per fare riferimento agli elementi della superclasse.
- **DB_VERSION:** versione del database, inizializzata a 1, variabile utilizzata, fondamentalmente per fare riferimento agli elementi della superclasse.
- **TABLE_C_M_AND_O_AND_A:** vero e proprio nome del database utilizzato per costruire le stringhe con le funzioni sql.

```
\\DataManager\\
[...]
public static final String TABLE_ROW_ID = "_id";
public static final String TABLE_ROW_C = "c";
public static final String TABLE_ROW_M = "m";
public static final String TABLE_ROW_O = "o";
public static final String TABLE_ROW_A = "a";
private static final String DB_NAME = "c_m_o_a_db";
private static final int DB_VERSION = 1;
private static final String TABLE_C_M_AND_O_AND_A = "c_m_and_o_and_a";
[...]
```

Come detto la struttura di questo database è molto simile a quella di **DataAppLoc**, possiamo ritrovare, infatti, le funzioni di costruzione del database.

```
\\DataManager\\
[...]
public DataManager (Context context){...}
[...]
public void onCreate(SQLiteDatabase db) {}
[...]
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {}
[...]
private class CustomSQLiteOpenHelper extends SQLiteOpenHelper{
  public CustomSQLiteOpenHelper(Context context){
  super(context, DB_NAME, null, DB_VERSION);
}
[...]
public void onCreate(SQLiteDatabase db){
  String newTableQueryString = "create table " + TABLE_C_M_AND_O_AND_A + " (" +
    TABLE_ROW_ID + " integer primary key autoincrement not null, " +
    TABLE_ROW_C + " text not null, " + TABLE_ROW_M + " text not null, " +
    TABLE_ROW_O + " text not null, " + TABLE_ROW_A + " text not null);";
            db.execSQL(newTableQueryString);
}
[...]
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){}
[...]
```

Oltre a queste abbiamo le funzioni che andremo ad utilizzare nelle varie activity:

- **insert:** permette di andare a salvare l'emento nel database, preparando la solita stringa da utilizzare nella funzione SQL.
- **delete:** permette di cancellare l'elemento facendo una **WHERE** sul nome della materia.
- **selectAll:** seleziona tutti gli elementi del database e ritorna il cursore con tutti i dati.
- **searchM:** ricerca la tabella del database il cui elemento "materia" corrisponda alla stringa *m*; anche questo torna un cursore che "punta" alla tabella trovata.

```
\\DataManager
[...]
public void insert(String m, String o, String a, String c){
  String query = "INSERT INTO " + TABLE_C_M_AND_O_AND_A + " (" +
    TABLE_ROW_C + ", " + TABLE_ROW_M + ", " + TABLE_ROW_O + ", " +
    TABLE_ROW_A + ") " + "VALUES (" + "'" + c + "'" + ", " + "'" + m + "'" +
    ", " + "'" + o + "'" + ", " + "'" + a + "'" + ");";
  Log.i("insert() = ", query);
  db.execSQL(query);
}
[...]
public void delete(String m){...}
[...]
public Cursor selectAll(){...}
[...]
public Cursor searchM(String m){...}
[...]
```

4 Implementazioni Xamarin

4.1 Login e Registrazione

4.2 Orario

4.3 I miei appunti

Dal **MainMenu**, scelta la voce miei appunti, viene avviata la classe **PreNotes**; qui viene visualizzata la lista delle materie inserite nel database tramite l'orario con la possibilità di aggiungerne di nuove.

La funzione **OnAppearing** permette di visualizzare elementi una volta inizializzata la classe, in questo caso viene caricata una **listView** contenente la lista delle materie presenti nel database **SubjectDatabase** tramite il metodo **GetSubjAsync** della classe SubjectDatabase; questo mi permette, appunto, di estrarre dal database tutte le materie salvate.

La funzione **OnListViewItemSelected** mi permette di attivare azioni catturando l'elemento selezionato dalla lista. In particolare alla pressione di un elemento viene chiamata la classe **Notes** passandogli come oggetto quello scelto.

Questa activity presenta anche la possibilità di aggiungere nuove materie al database, infatti è presente nella navbar un bottone per aggiungere materie, questo scatena lal funzione **OnAggMateriaClicked** che lancia appunto l'activity per aggiungere una materia, **AddSubj**.

```
\\PreNotes\\
[...]
protected override async void OnAppearing() {
  base.OnAppearing();
  listView.ItemsSource = await App.SubjectsDatabase.GetSubjAsync();
}
[...]
async void OnListViewItemSelected(object sender,
  SelectedItemChangedEventArgs e) {
  if(e.SelectedItem != null) {
    await Navigation.PushAsync(new Notes { BindingContext = e.SelectedItem
      as Subjects });
  }
}
[...]
async void OnAggMateriaClicked(object sender, EventArgs e) {
  await Navigation.PushAsync(new AddSubj { BindingContext = new Subjects() });
}
[...]
```

Se si scegli di aggiungere una nuova materia ci si trova davanti ad una edit text, ed un bottone per salvare. All'interno della classe si utilizza una funzione per catturare l'evento sul bottone salva; all'interno di questa viene utilizzata una variabile stringa, *materia*, e le viene assegnata il valore estratto dalla "editText" nella quale andrebbe inserito il nome della materia da aggiungere. Successivamente viene istanziato un oggetto della classe Subject, *control*; grazie alla funzione **ControlSubjAsync** della classe SubjectsDatabase, viene banalmente controllato se l'elemento esisteva già nel database o meno; in caso positivo, primo *if* non viene salvata, nel caso in cui la materia non sia già nel database, questa viene salvata, lanciando la sunzione **SaveSubjAsync** della solita classe SubjectsDatabase. Terminata questa routine con una **PopAsync** si torna all'activity precedente.

```
\\AddSubj\\
[...]
async void OnSaveButtonClicked(object sender, EventArgs e) {
[...]
  string materia = subj.Subject.ToString();
  Subjects control = new Subjects();
  control = await App.SubjectsDatabase.ControlSubjAsync(materia);
  if (control != null) {...}
  else {
    await App.SubjectsDatabase.SaveSubjAsync(subj);
  }
  await Navigation.PopAsync();
}
[...]
```

Tornati alla lista delle materie, operiamo una scelta, a questo punto, come già detto, verrà lanciata l'activity **Notes**.

Nella classe Notes viene visualizzata, nuovamente grazie alla funzione **OnAppearing**, la lista degli appunti relativi alla materia, in particolare viene estratto dall'oggetto in **BindingContext** la stringa contenente la materia e messo nella variabile *mat*, sunque, può partire la ricerca nel database degli appunti **NoteDatabase** utilizzado come chiave il nome della materia; questo ci permette di visualizzare una lista con gli appunti relativi alla materia scelta.

L'activity include la possibilità di aggiungere un appunto, tramite il bottone nella navbar, infatti, viene scatenata la funzione **OnNoteAddedClicked**, questa prende nuovamente il nome della materia in questione e lancia l'activity **NotePage** passandole come parametro il nome della materia.

Scelto l'appunto da aprire la funzione **OnListViewItemSelected** permette di catturare i dati relativi all'oggetto scelto e lancia nuovamente l'activity NotePage, passando, stavolta, come contesto l'oggetto scelto.

```
\\Notes\\
[...]
protected override async void OnAppearing() {
  [...]
  var temp = (Subjects)BindingContext;
  string mat = temp.Subject.ToString();
  listView.ItemsSource = await App.NoteDatabase.GetNoteBySubj(mat);
}
[...]
async void OnNoteAddedClicked(object sender, EventArgs e) {
  var temp = (Subjects)BindingContext;
  string mat = temp.Subject.ToString();
  await Navigation.PushAsync(new NotePage(mat));
}
[...]
async void OnListViewItemSelected(object sender,
  SelectedItemChangedEventArgs e) {
  if(e.SelectedItem != null){
    await Navigation.PushAsync(new NotePage { BindingContext =
      e.SelectedItem as Note });
  }
}
[...]
```

La classe **NotePage** ci permette di visualizzare salvare o eliminare un appunto.

All'apertura vengono inizializzate diverse editText che contengono dati relativi a titolo data e l'appunto stesso. Nel caso in cui venga attivato il bottone per il salvataggio viene creato un oggetto *note* e catturate le stringhe nelle edit, queste vengono, poi, assegnate ai campi dell'oggetto *note*, fatto ciò viene salvato l'oggetto con la funzione **SaveNoteAsync** e chiusa l'activity.

Se, invece, si sceglie di eliminare un appunto viene attivata la funzione **OnDeleteButtonClicked**, questa, utilizza il metodo **DeleteNoteAsync** passando come argomento l'intero oggetto *note*, in tal modo verrà cancellato tutto ciò che è relativo all'appunto in questione.

```
\\NotePage\\
[...]
async void OnSaveButtonClicked(object sender, EventArgs e) {
  Note note = new Note();
  string title = this.EdtTitle.Text;
  string date = this.EdtDate.Text;
  string notes = this.EdtNotes.Text;
  note.Subject = mat;
  note.Title = title;
  note.Date = date;
  note.Notes = notes;
  await App.NoteDatabase.SaveNoteAsync(note);
  await Navigation.PopAsync();
  [...]
}
[...]
async void OnDeleteButtonClicked(object sender, EventArgs e) {
  var note = (Note)BindingContext;
```

```
19    await App.NoteDatabase.DeleteNoteAsync(note);
20    await Navigation.PopAsync();
21  }
22  [...]
```

4.4 Appunti condivisi

4.5 Impostazioni

4.6 Database

Ciò che è inerente al funzionamento dei database è contenuto nelle cartelle **Data** e **Models**. Prendiamo come riferimento le classi **Note**, **Subjects**, **SubjectsDatabase** e **NotesDatabase**, contenute rispettivamente nelle cartelle Models e Data. La creazione di questi database avviene al lancio dell'applicazione, infatti nella classe **App** possiamo notare la deefinizione di funzioni statiche per la creazione dei database. Viene riportato il codice relativo al database utile ad ospitare le materie, ma allo stesso modo vengono implementati gli altri.

```
1  \\App\\
2  [...]
3  public static SubjectsDatabase SubjectsDatabase {d
4    get {
5    if (databaseSubjects == null) {
6      databaseSubjects = new SubjectsDatabase(Path.Combine
7        (Environment.GetFolderPath(Environment.SpecialFolder.
8        LocalApplicationData), "subjects.db3"));
9      }
10   return databaseSubjects;
11   }
12 }
13 [...]
```

Le prime due riguardano fondamentalmente la struttura delle tabelle del database. Al loro interno sono, infatti, definite stringhe quali materia, titolo, data, e appunto, queste definiscono, grazie all'importazione del pacchetto **SQLite**, funzioni **get** e **set**.

```
1  \\Note\\
2  [...]
3  public class Note {
4    [PrimaryKey, AutoIncrement]
5    public int ID { get; set; }
6    public string Subject { get; set; }
7    public string Title { get; set; }
8    public string Date { get; set; }
9    public string Notes { get; set; }
10 }
11 [...]
```

```
1  \\Subjects\\
2  [...]
3  public class Subjects {
4    [PrimaryKey, AutoIncrement]
5    public int ID { get; set; }
6    public string Subject { get; set; }
7  }
8  [...]
```

Per quanto riguarda le altre due classi, al loro interno abbiamo funzioni per utilizzare e manipolare dati nel database.

In particolare per quanto riguarda il database relativo alle materie abbiamo:

- **GetSubjAsync:** questo viene utilizzato per estrarre tutte le materie dal database.
- **SaveSubjAsync:** prende in argomento unoggetto subj costruito riempiendo i suoi campi definiti nella classe Subjects.
- **DeleteSubjAsync:** utilizzata per eliminare un oggetto dal database, anche questo metodo prende come parametro l'oggetto che si desidera cancellare.

- **ControlSubjAsync:** funzione utilizzata per controllare se la materia che si desidera salvare è già presente nel database, infatti prende come argomento una stringa contenente appunto il nome della materia.

Tutti questi metodo sono di tipo **Task**< ... >, questo mi permette di utilizzare come ritorno della funzione tabelle o comunque elementi del database.

```
\\SubjectsDatabase\\
[...]
public Task<List<Subjects>> GetSubjAsync() {...}
[...]
public Task<int> SaveSubjAsync(Subjects subject) {...}
[...]
public Task<int> DeleteSubjAsync(Subjects subject) {...}
[...]
public Task<Subjects> ControlSubjAsync(string subject) {...}
[...]
```

In modo del tutto analogo è composta la classe **NotesDatabase**.