BUTE Department of Automation and Applied Informatics
AMORG
APPLIED MOBILE RESEARCH GROUP

Android alapú szoftverfejlesztés

6. Labor

SQLite használata, fájlkezelés Android platformon

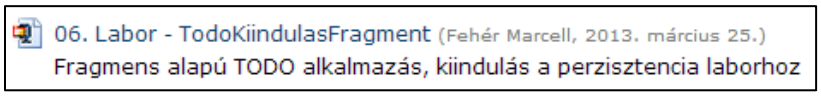
Tartalom

1 Felkészülés a laborra

A labor célja a relációs adatbáziskezelés bemutatása, aminek szemléltetésére a korábban elkészített Todo lista alkalmazást fejlesztjük tovább. A labor során SQLite adatbázisban fogjuk tárolni a Todo elemeinket, így azok nem fognak elveszni, ha a felhasználó elnavigál az alkalmazásunkból, vagy elforgatja azt.

2 Kiinduló projekt

Töltsük le a kezdeti alkalmazást letölthető a tárgy honlapjáról.



https://www.aut.bme.hu/Course/android

Ez a Todo alkalmazás azon verziója, amely már *Fragment*-ekkel működik, azonban nem tárolja perzisztens módon el a létrehozott Todo objektumokat.

Indítsuk el az fejlesztő környezetet, zárjuk be az összes megnyitott projektet majd importáljuk be a letöltött zip-et teljes projektként (*Package Explorer* üres területen jobb gomb → Import → *General/Existing Projects into Workspace* →***Select archive file*** → Browse → Finish)

3 Adattárolás SQLite adatbázisban

Célunk, hogy a memóriában tárolás helyett a Todo objektumaink egy SQLite adatbázisban legyenek perzisztensen mentve, így azok nem vesznek el kilépéskor sem. A *TodoAdapter* ezek után nem a memóriában tárolt listát fogja megjeleníteni, hanem az adatbázist olvassa és köti össze a *ListView*-al.

A feladat megvalósítása három részből áll:

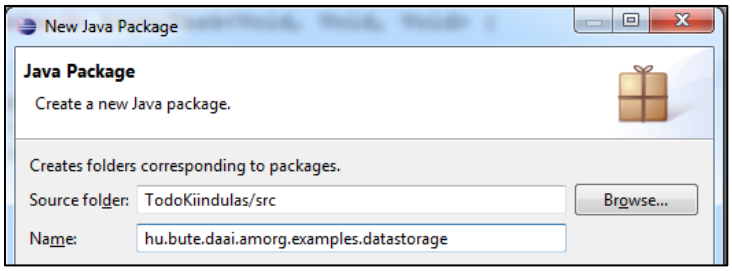
1. Todo objektumok tárolása adatbázisban
2. Listával dolgozó adapter átalakítása adatbázissal (*Cursor*-ral) működővé
3. Workflow átírása

3.1 Todo-k tárolása adatbázisban

Az SQLite adatbáziskezelő használatához segítséget nyújt a platform, mégpedig az *SQLiteOpenHelper* osztállyal. Ebből származtatva olyan saját osztályt hozhatunk létre,

ami referenciát szolgáltat az általunk használt teljes adatbázisra, így tehát több entitás osztály és adatbázis tábla esetén is elég egy ilyen segédosztály.

Hozzuk létre a projektben egy új package-et (*src*-n belül New→Package). A csomag neve legyen „**`hu.bute.daai.amorg.examples.datastorage`**”



Ezen belül hozzunk létre egy új osztályt *DatabaseHelper* néven, melyek ősosztálya az *SQLiteOpenHelper*. Tartalma (másoljuk át a patch.txt-ből, hogy a paraméterek neve egyezzen!):

```
public class DatabaseHelper extends SQLiteOpenHelper {

	public DatabaseHelper(Context context, String name, CursorFactory
factory, int version) {
		super(context, name, factory, version);
		// TODO Auto-generated constructor stub
	}

	@Override
	public void onCreate(SQLiteDatabase db) {
		// TODO Auto-generated method stub
	}

	@Override
	public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
		// TODO Auto-generated method stub
	}
}
```

Rakjuk rendben az importált osztályokat a Ctrl+Shift+O kombinációval.

Az *SQLiteOpenHelper* osztályból történő származtatás a konstruktoron kívül két metódus kötelező felüldefiniálását írja elő. Az *.onCreate()*-ben kell futtatnunk a sémalétrehozó SQL szkriptet, míg az *.onUpgrade()*-ben kezelhetjük le az adatbázis verzióváltásával kapcsolatos feladatokat. Legegyszerűbb esetben itt töröljük, majd újra létrehozzuk az egész sémát, de ekkor az adatokat is elveszítjük.

Az adatbáziskezelés során sok konstans jellegű változóval kell dolgoznunk, mint például a táblákban lévő oszlopok nevei, táblák neve, adatbázis fájl neve, séma létrehozó és törlő szkiptek, stb. Ezeket érdemes egy közös helyen tárolni, így szerkesztéskor vagy új entitás

bevezetésekor nem kell a forrásfájlok között ugrálni, valamint egyszerűbb a teljes adatbázist létrehozó és törlő szkripteket generálni. Hozzunk létre egy új osztályt a *datastorage* csomagban *DbConstants* néven, és statikus tagváltozóként vegyünk fel minden szükséges konstanst:

```
public class DbConstants {

	// Broadcast Action, amely az adatbazis modosulasat jelzi
	public static final String ACTION_DATABASE_CHANGED =
	"hu.bute.daai.amorg.examples.DATABASE_CHANGED";

	// fajlnev, amiben az adatbazis lesz
	public static final String DATABASE_NAME = "data.db";
	// verzioszam
	public static final int DATABASE_VERSION = 1;
	// osszes belso osztaly DATABASE_CREATE szkriptje osszefuzve
	public static String DATABASE_CREATE_ALL = Todo.DATABASE_CREATE;
	// osszes belso osztaly DATABASE_DROP szkriptje osszefuzve
	public static String DATABASE_DROP_ALL = Todo.DATABASE_DROP;

	/* Todo osztaly DB konstansai */
	public static class Todo{
		// tabla neve
		public static final String DATABASE_TABLE = "todo";
		// oszlopnevek
		public static final String KEY_ROWID = "_id";
		public static final String KEY_TITLE = "title";
		public static final String KEY_PRIORITY = "priority";
		public static final String KEY_DUEDATE = "dueDate";
		public static final String KEY_DESCRIPTION = "description";
		// sema letrehozo szkript
		public static final String DATABASE_CREATE =
		    "create table if not exists "+DATABASE_TABLE+" ( "
		    + KEY_ROWID +" integer primary key autoincrement, "
		    + KEY_TITLE + " text not null, "
		    + KEY_PRIORITY + " text, "
		    + KEY_DUEDATE +" text, "
		    + KEY_DESCRIPTION +" text"
		    + "); ";
		// sema torlo szkript
		public static final String DATABASE_DROP =
			"drop table if exists " + DATABASE_TABLE + "; ";
	}
}
```

Figyeljük meg hogy a *DbConstants* osztályon belül létrehoztunk egy statikus belső *Todo* nevű osztályt, amiben a *Todo* entitásokat tároló tábla konstansait tároljuk. Amennyiben az alkalmazásunk több entitást is adatbázisban tárol (gyakori eset!), akkor érdemes az egyes osztályokhoz tartozó konstansokat külön-külön belső statikus osztályokban tárolni. Így sokkal átláthatóbb és karbantarthatóbb lesz a kód, mintha ömlesztve beírnánk a *DbConstants*-ba az összes tábla összes konstansát. Ezek a belső osztályok praktikusan ugyanolyan névvel léteznek mint az entitás osztályok (jelen esetben mindkettő neve

Todo) , azonban mivel más package-ben vannak ez nem okoz fordítási hibát (részben ezért is csináltuk a *datastorage* végű csomagot).

Ha megvannak a konstansok, írjuk meg a *DatabaseHelper* osztály metódusait. A konstruktor paraméterei közül törölhetjük a *CursorFactory*-t és a verziószámot, és az ősosztály konstruktorának hívásakor a megfelelő helyeken adjunk át null-t (így a default *CursorFactory*-t fogja használni), valamint a `DbConstants.DATABASE_VERSION` stringet verzióként. Az *onCreate()* és *onUpgrade()* metódusokban használjuk a *DbConstants*-ban ebből a célból létrehozott konstansokat.

```
public class DatabaseHelper extends SQLiteOpenHelper {

	public DatabaseHelper(Context context, String name) {
		super(context, name, null, DbConstants.DATABASE_VERSION);
	}

	@Override
	public void onCreate(SQLiteDatabase db) {
		db.execSQL(DbConstants.DATABASE_CREATE_ALL);
	}

	@Override
	public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
		db.execSQL(DbConstants.DATABASE_DROP_ALL);
		db.execSQL(DbConstants.DATABASE_CREATE_ALL);
	}
}
```

A séma létrehozásához és megnyitásához szükséges osztályok rendelkezésre állnak, a következő feladatunk az entitás osztályok felkészítése az adatbázisból történő használatra. Ehhez meg kell írnunk azokat a kódrészleteket, melyek egy memóriába lévő Todo objektumot képesek adatbázisba írni, onnan visszaolvasni, módosítani valamint törölni (természetesen más funkciók is szükségesek lehetnek). Ezt a kódot az entitás osztály (Todo) helyett érdemes egy külön osztályban megvalósítani a *datastorage* végű csomagon belül (*TodoDbLoader*):

```
public class TodoDbLoader {

	private Context ctx;
	private DatabaseHelper dbHelper;
	private SQLiteDatabase mDb;

	public TodoDbLoader(Context ctx) {
		this.ctx = ctx;
	}

	public void open() throws SQLException{
		// DatabaseHelper objektum
		dbHelper = new DatabaseHelper(
```

```
                        ctx, DbConstants.DATABASE_NAME);
            // adatbázis objektum
            mDb = dbHelper.getWritableDatabase();
            // ha nincs még séma, akkor létrehozzuk
            dbHelper.onCreate(mDb);
      }

      public void close(){
            dbHelper.close();
      }
      // CRUD és egyéb metódusok
}
```

A létrehozó, módosító és törlő metódusok (*TodoDbLoader*-en belül!):

```
// INSERT
public long createTodo(Todo todo){
      ContentValues values = new ContentValues();
      values.put(DbConstants.Todo.KEY_TITLE, todo.getTitle());
      values.put(DbConstants.Todo.KEY_DUEDATE, todo.getDueDate());
      values.put(DbConstants.Todo.KEY_DESCRIPTION,
todo.getDescription());
      values.put(DbConstants.Todo.KEY_PRIORITY,
todo.getPriority().name());

      return mDb.insert(DbConstants.Todo.DATABASE_TABLE, null, values);
}

// DELETE
public boolean deleteTodo(long rowId){
      return mDb.delete(
                  DbConstants.Todo.DATABASE_TABLE,
                  DbConstants.Todo.KEY_ROWID + "=" + rowId,
                  null) > 0;
}

// UPDATE
public boolean updateProduct(long rowId, Todo newTodo){
      ContentValues values = new ContentValues();
      values.put(DbConstants.Todo.KEY_TITLE, newTodo.getTitle());
      values.put(DbConstants.Todo.KEY_DUEDATE, newTodo.getDueDate());
      values.put(DbConstants.Todo.KEY_DESCRIPTION,
newTodo.getDescription());
      values.put(DbConstants.Todo.KEY_PRIORITY,
newTodo.getPriority().name());
      return mDb.update(
                  DbConstants.Todo.DATABASE_TABLE,
                  values,
                  DbConstants.Todo.KEY_ROWID + "=" + rowId ,
                  null) > 0;
}
```

Fussuk át és értelmezzük a bemásolt kódot.

A kód bemásolása után importáljuk be a megfelelő osztályok. A Todo osztály két helyen is szerepel, egyrészt entitás osztályként (a *.data* végű csomagban), másrészt az adatbázis konstansokat tároló belső osztályként (a *.datastorage* végű csomagban). A bemásolt kód célja, hogy a Todo entitásokat hozza létra, módosítsa vagy törölje az adatbázisban, így az entitás osztályt importáljuk be (*hu.bute.daai.amorg.examples.**data.Todo***).

Általában igaz, hogy szükséges olyan metódusok megírása, melyek képesek egy rekordot visszaadni, valamint egy kurzor objektummal visszatérni, ami a teljes rekord halmazra mutat. Ezek implementációja (még mindig a *TodoDbLoader* osztályban):

```
// minden Todo lekérése
public Cursor fetchAll(){
	// cursor minden rekordra (where = null)
	return mDb.query(
			DbConstants.Todo.DATABASE_TABLE,
			new String[]{
					DbConstants.Todo.KEY_ROWID,
					DbConstants.Todo.KEY_TITLE,
					DbConstants.Todo.KEY_DESCRIPTION,
					DbConstants.Todo.KEY_DUEDATE,
					DbConstants.Todo.KEY_PRIORITY
			}, null, null, null, null,
DbConstants.Todo.KEY_TITLE);
}

// egy Todo lekérése
public Todo fetchTodo(long rowId){
	// a Todo-ra mutato cursor
	Cursor c = mDb.query(
			DbConstants.Todo.DATABASE_TABLE,
			new String[]{
					DbConstants.Todo.KEY_ROWID,
					DbConstants.Todo.KEY_TITLE,
					DbConstants.Todo.KEY_DESCRIPTION,
					DbConstants.Todo.KEY_DUEDATE,
					DbConstants.Todo.KEY_PRIORITY
			}, DbConstants.Todo.KEY_ROWID + "=" + rowId,
			null, null, null, DbConstants.Todo.KEY_TITLE);
	// ha van rekord amire a Cursor mutat
	if(c.moveToFirst())
		return getTodoByCursor(c);
	// egyebkent null-al terunk vissza
	return null;
}
```

A kód bemásolása és importok rendezése után hibát kapunk a forrás végénél a *getTodoByCursor(c)* metódushívásra. Ez még valóban nincs implementálva, hamarosan pótoljuk.

3.2 Lista feltöltése adatbázisból

Ha az Adapter osztályunkat adatbázisból szeretnénk feltölteni, akkor a *BaseAdapter* helyett a *CursorAdapter* ősosztályból kell származtatnunk. Ez a konstruktor megírásán kívül két metódus implementációját írja elő kötelezően. A *newView(Context context, Cursor cursor, ViewGroup parent)* létrehoz egy sornak megfelelő *View*-t, amit feltölt adatokkal és visszatér vele (ugyanaz a funkció mint *BaseAdapter* esetén a *getView()*). A takarékos erőforrás felhasználás érdekében az adatfeltöltést érdemes a másik kötelező metódusban, a *bindView(View view, Context context, Cursor cursor)* -ban végezni. A forráskód a következő (ne felejtsük el átírni a *BaseAdapter*-ből származtatást *CursorAdapter*-re!):

```
public class TodoAdapter extends CursorAdapter {

	public TodoAdapter(Context context, Cursor c) {
		super(context, c, false);
	}

	@Override
	public View newView(Context context, Cursor cursor, ViewGroup
parent) {
		final LayoutInflater inflater =
LayoutInflater.from(context);
		View row = inflater.inflate(R.layout.todorow, null);
		bindView(row, context, cursor);
		return row;
	}

	// UI elemek feltöltése
	@Override
	public void bindView(View view, Context context, Cursor cursor) {
		// referencia a UI elemekre
		TextView titleTV = (TextView)
view.findViewById(R.id.textViewTitle);
		TextView dueDateTV = (TextView)
view.findViewById(R.id.textViewDueDate);
		ImageView priorityIV = (ImageView)
view.findViewById(R.id.imageViewPriority);

		// Todo példányosítás Cursorból
		Todo todo = TodoDbLoader.getTodoByCursor(cursor);

		// UI elemek
		titleTV.setText(todo.getTitle());
		dueDateTV.setText(todo.getDueDate());
		switch (todo.getPriority()) {
			case HIGH:
				priorityIV.setImageResource(R.drawable.high);
				break;
			case MEDIUM:
				priorityIV.setImageResource(R.drawable.medium);
```

```
                        break;
                    case LOW:
                        priorityIV.setImageResource(R.drawable.low);
                        break;
                }

        }


        @Override
        public Todo getItem(int position) {
                getCursor().moveToPosition(position);

                return TodoDbLoader.getTodoByCursor(getCursor());
        }

}
```

Az implementáció hivatkozik a *TodoLoader* még nem létező statikus *getTodoByCursor(Cursor)* metódusára, ami egy rekordra állított *Cursor*-t kap paraméterként, és egy Todo objektummal tér vissza. Ennek kódja triviális (visszatér egy adatbázisból lekért adatokkal példányosított Todo objektummal). Írjuk meg a metódust a a ***TodoDbLoader*** osztály végén.

```
public static Todo getTodoByCursor(Cursor c){
        return new Todo(

        c.getString(c.getColumnIndex(DbConstants.Todo.KEY_TITLE)), //
title

        Priority.valueOf(c.getString(c.getColumnIndex(DbConstants.Todo.KE
Y_PRIORITY))), // priority

        c.getString(c.getColumnIndex(DbConstants.Todo.KEY_DUEDATE)), //
dueDate

        c.getString(c.getColumnIndex(DbConstants.Todo.KEY_DESCRIPTION))
// description
                    );
}
```

Mentés (és a Prority osztály importálása) után a hibák eltűnnek a *TodoAdapter* és a *TodoDbLoader* osztályokból (viszont megjelent a *TodoListFragment*-ben, mivel változott az adapter konstruktorának paraméterezése a BaseAdapter→CursorAdapter áttéréskor).

3.3 Egyedi Application objektum készítése

Készítsünk egy egyedi *Application* osztályt, *TodoApplication* néven. Az *Application* az alkalmazás futása során folyamatosan jelen lévő objektum, melyet a futtatókörnyezet hoz létre automatikusan, élettartama az alkalmazáshoz tartozó process élettartamához

köthető. A *TodoApplication* fogja fenntartani az adatbáziskapcsolatot reprezentáló *TodoDbLoader* objektumot, így nem kell minden Activity-ből külön-külön, erőforrás pazarló módon példányosítani.

Hozzunk létre tehát egy *.application* package-et, majd abban készítsük el a *TodoApplication* osztályt:

```
public class TodoApplication extends Application {
	private static TodoDbLoader dbLoader;

	public static TodoDbLoader getTodoDbLoader() {
		return dbLoader;
	}

	@Override
	public void onCreate() {
		super.onCreate();

		dbLoader = new TodoDbLoader(this);
		dbLoader.open();
	}

	@Override
	public void onTerminate() {
		// Close the internal db
		dbLoader.close();

		super.onTerminate();
	}
}
```

A *AndroidManifest*-ben meg kell adnunk, hogy a rendszer melyik osztályt példányosítsa induláskor, mint *Application* objektum. Ehhez az xml-ben lévő <application> node-ban fel kell vennünk egy új attribútumot android:name néven, és beállítani az újonnan elkészített saját Application példányunk minősített (*fully-qualified*) osztálynevét:

```
....
    <application

android:name="hu.bute.daai.amorg.examples.application.TodoApplication"
        android:icon="@drawable/icon"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
....
```

Ezt követően a *TodoApplication* osztály *getTodoDbLoader()* statikus metódusával bármikor elérhetjük az SQLite adatbáziskapcsolatunkat.

3.4 TodoListFragment módosítása

Az adatbázis perzisztencia használatához minden rendelkezésre áll, már csak az *TodoListFragment*-ben kell áttérnünk listáról SQL-re.

Módosítsuk a *TodoListFragment* osztály elején a mezőket az alábbi módon:

```
public class TodoListFragment extends ListFragment implements
            ITodoCreateFragment {

      // Log tag
      public static final String TAG = "TodoListFragment";

      // State
      private TodoAdapter adapter;
      private LocalBroadcastManager lbm;

      // Listener
      private ITodoListFragment listener;

      // DBloader
      private TodoDbLoader dbLoader;
      private GetAllTask getAllTask;

      ...
```

Az diszk I/O műveleteket javasolt külön szálon, aszinkron módon futtatni, mivel könnyen megakaszthatják a UI szálat lassú lefutásukkal. Készítsünk a *TodoListFragment* osztályban egy *AsyncTask* osztályból származó **belső osztályt** *GetAllTask* néven, amely aszinkron módon kérdezi le az adatbázisunktól az összes elmentett Todo rekordot:

```
private class GetAllTask extends AsyncTask<Void, Void, Cursor> {

            private static final String TAG = "GetAllTask";

            @Override
            protected Cursor doInBackground(Void... params) {
                  try {
                        Cursor result = dbLoader.fetchAll();

                        if (!isCancelled()) {
                              return result;
                        } else {
                              Log.d(TAG, "Cancelled, closing cursor");
                              if (result != null) {
                                    result.close();
                              }

                              return null;
                        }
                  } catch (Exception e) {
```

```
                        return null;
                  }
            }

            @Override
            protected void onPostExecute(Cursor result) {
                  super.onPostExecute(result);

                  Log.d(TAG, "Fetch completed, displaying cursor
results!");
                  try {
                        if (adapter == null) {
                              adapter = new TodoAdapter(getActivity()
                                          .getApplicationContext(),
result);
                              setListAdapter(adapter);
                        } else {
                              adapter.changeCursor(result);
                        }

                        getAllTask = null;
                  } catch (Exception e) {
                  }
            }

      }
```

Készítsünk egy helyi *BroadcastReceiver* osztályt (ezt is a *TodoListFragment*-en belül!), amely figyel azokra a Broadcast-üzenetekre, melyek az adatbázis módosulását jelzik. Erre az eseményre a *BroadcastReceiver* osztályunk a Todo-lista tartalmának újbóli lekérdezésével fog reagálni (az ehhez tartozó hiányzó *.refreshList()* metódust később készítjük el):

```
private BroadcastReceiver updateDbReceiver = new BroadcastReceiver() {
      @Override
      public void onReceive(Context context, Intent intent) {
            refreshList();
      }
};
```

Ezt követően a *Fragment* életciklus-hívásait az alábbi módon adjuk meg:

Az *.onCreate()* hívásban kérjünk referenciát a Support Library-ben található *LocalBroadcastManager* osztály példányára, illetve kérjük el a referenciát a *TodoApplication* objektumunktól a nyitott adatbázis kapcsolatot tartalmazó *TodoDbLoader*-re.

```
@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
```

```
        setHasOptionsMenu(true);

        lbm = LocalBroadcastManager.getInstance(getActivity());
        dbLoader = TodoApplication.getTodoDbLoader();
}
```

Az *.onStart()* hívásban csupán közöljük a keretrendszerrel, hogy a listánk támogatja a Context menüt:

```
@Override
public void onStart() {
        super.onStart();

        registerForContextMenu(getListView());
}
```

Az *.onResume()* hívásban (még nincs implementálva) regisztráljuk be a korábban létrehozott BroadcastReceiver osztálypéldányunkat, illetve rögtön kérjük a Todo-lista tartalmának lekérdezését:

```
@Override
public void onResume() {
        super.onResume();

        // Kódból regisztraljuk az adatbazis modosulasara figyelmezteto
        Receiver-t
        IntentFilter filter = new IntentFilter(
                        DbConstants.ACTION_DATABASE_CHANGED);
        lbm.registerReceiver(updateDbReceiver, filter);

        // Frissitjuk a lista tartalmat, ha visszater a user
        refreshList();
}
```

Az *.onPause()* hívásban (ez sem létezik még) kiregisztráljuk a BroadcastReceiver-ünket, illetve ha van éppen folyamatban DB-lekérdezés, akkor azt megszakítjuk:

```
@Override
public void onPause() {
        super.onPause();

        // Kiregisztraljuk az adatbazis modosulasara figyelmezteto
        Receiver-t
        lbm.unregisterReceiver(updateDbReceiver);

        if (getAllTask != null) {
                getAllTask.cancel(false);
        }
}
```

Az *.onDestroy()* hívásban (szintén nincs még) bezárjuk az adapterhez csatolt, Todo elemeket tartalmazó *Cursor* objektumot, ha létezik ilyen:

```
@Override
public void onDestroy() {
	super.onDestroy();

	// Ha van Cursor rendelve az Adapterhez, lezarjuk
	if (adapter != null && adapter.getCursor() != null) {
		adapter.getCursor().close();
	}
}
```

Végül készítsük el a hiányzó *.refreshList()* metódust is, amely elindítja az adatbázist lekérdező aszinkron folyamatot:

```
private void refreshList() {
	if (getAllTask != null) {
		getAllTask.cancel(false);
	}

	getAllTask = new GetAllTask();
	getAllTask.execute();
}
```

A kódunkban már csupán egy hibás rész van, az *.onTodoCreated()* callback metódus törzsében. Egyelőre ezt kommentezzük ki, a következő lépésben valósítjuk majd meg a helyes működést.

Ha minden kódrészlet a megfelelő helyen van, akkor az alkalmazás indulásakor ugyanaz a felhasználói felület jelenik meg, mint a kiinduló projekt esetén, azonban most még üres listát látunk, hiszen üres az adatbázis. Ahhoz, hogy működjön a Todo-elem létrehozása funkció újra, módosítanunk kell az *.onTodoCreated(...)* callbacket a következő módon:

```
// ITodoCreateFragment
public void onTodoCreated(Todo newTodo) {
	dbLoader.createTodo(newTodo);

	refreshList();
}
```

Így már képesek vagyunk új Todo elemeket létrehozni. Próbáljuk ki, hogy immár forgatás után, illetve elnavigálás után is megmaradnak a létrehozott Todo-k.
Az eredeti funkcionalitás további részeinek megvalósítása önálló feladatként oldandók meg.

Kötelező feladatok:

- ContextMenu átalakítása (kiválasztott Todo elem törlése)

- Összes Todo elem törlése funkció (pl. Options menüből elérve)

Jó munkát!