

# Mobil programozási alapok

Dokumentáció

Danyi Kristóf Milán

GQOKMW

Taskmanager mobilalkamazás

## A mobilalkalmazás célja

A Taskmanager egy olyan mobilalkalmazás, amelyben a teendőinket tudjuk rögzíteni, és később visszanézni azt.

Ismert közmondás az “Akinek nincs esze legyen notesze”. Ezt a feledékeny embereknek szokták mondani, és ebben segít ez az alkalmazás, hogy a teendőinket amit nem azonnal teszünk meg fel tudjuk jegyzetelni

## Az alkalmazás előnye

Az alkalmazásunk akár lenne egy sima Notepad alkalmazás is, egy jegyzettömb, mint ahogy a mondás is tartja, de az előnye ennek a mobilalkalmazásnak, hogy itt egy képernyőn külön tudjuk szedni a teendőket, ezeknek a teendőknek egy leírást adni, hogyha a címéből nem lenne egyértelmű.

Lehetőségünk van prioritást adni ezeknek a teendőknek, így a listában a fontosabb dolgaink felül jelennek meg.

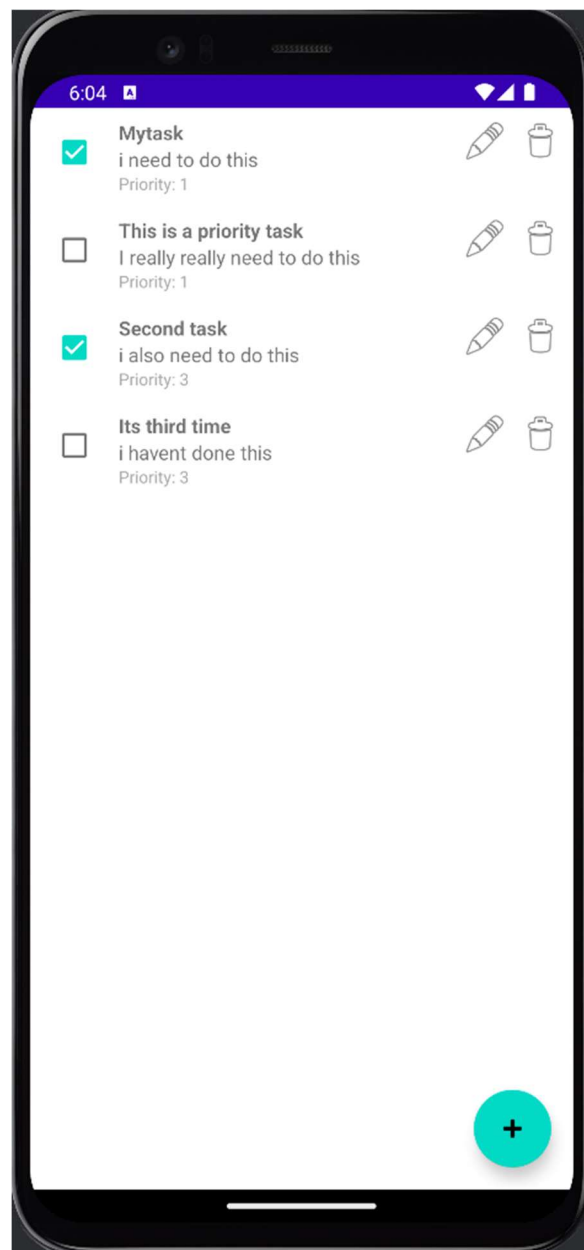
Emellett a teendőket késznek is meg tudjuk jelölni, abban az esetben ha egy nap összegezni szeretnénk, hogy mit sikerült elérnünk, persze ez nem jelenti azt hogy nem tudjuk törölni a teendőket.

Amennyiben csak egy picit változik valamennyi teendőnk, vagy valamit véletlenül rosszul konfiguráltunk, lehetőségünk van ezeket a teendőket az alkalmazásban módosítani is.

# Az alkalmazás funkciói

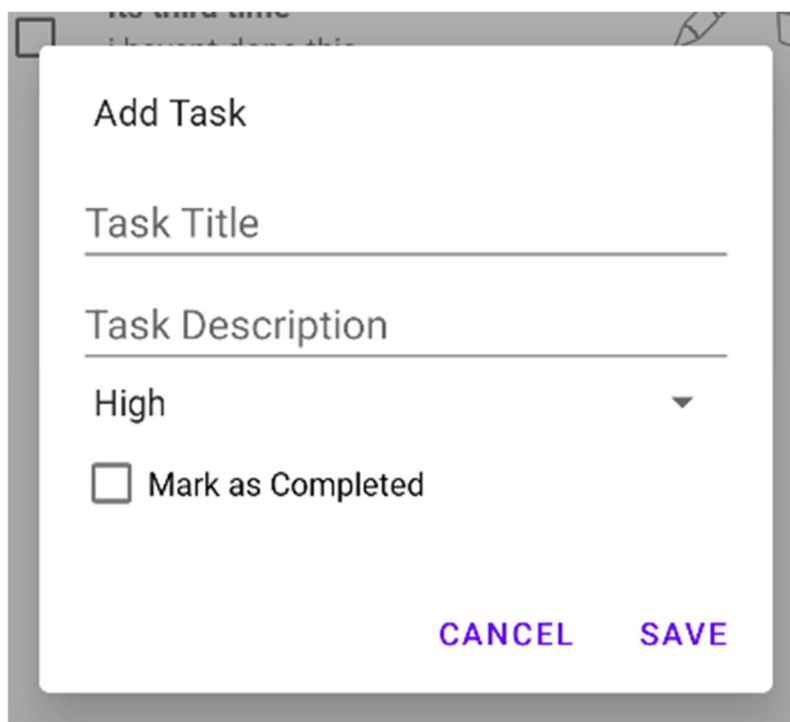
## Főképernyő

Az alkalmazás első megnyitásakor Egy képernyő jön velünk szembe, ahol fel vannak sorolva a teendőink. Itt lehetőségünk van új teendőt hozzáadni, a meglévő teendőinket módosítani, törölni őket, valamint késznek jelölni őket



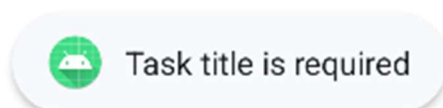
## Teendő hozzáadása

A jobb alsó sarokban található „+”-ra nyomva egy ablak jön fel, ahol részleteket kér az új teendőről amit hozzáadni kívánunk a listánkhoz

A screenshot of a mobile application's 'Add Task' dialog box. The dialog is white with a gray border. It contains the following elements: a title 'Add Task', a text input field for 'Task Title', another text input field for 'Task Description', a dropdown menu currently showing 'High', and a checkbox labeled 'Mark as Completed'. At the bottom right, there are two buttons: 'CANCEL' and 'SAVE', both in purple text.

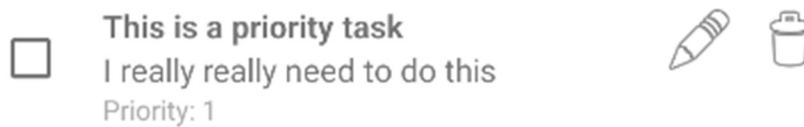
Itt egy kis címet adhatunk a teendőnknek, egy leírást, kiválaszthatjuk a prioritását, valamint ha csak jegyzetelni akarjuk hogy mit értünk mostanában el, egyből késznek is tekinthetjük.

Egy kis felhasználói védelemként ha nem írunk címet, akkor figyelmeztet minket az alkalmazás, és nem veszi fel a teendőt.



## Teendő módosítása és törlése

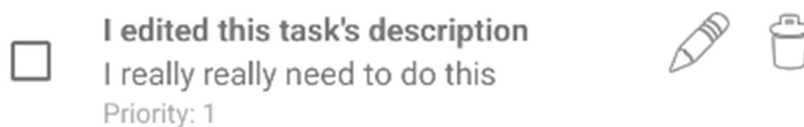
A teendőinket a teendő jobb oldalántalálható ceruza és kuka ikonnal módosíthatjuk valamint törölhetjük



A törlés funkció egyértelmű, a módosítás pedig egy létrehozás ablakot dob fel, amelyen a teendő adatait láthatjuk.

















A modal dialog box titled "Edit Task". It contains the following fields: a text input with "This is a priority task", another text input with "I really really need to do this", and a dropdown menu currently showing "High". At the bottom left is a checkbox labeled "Mark as Completed". At the bottom right are two buttons: "CANCEL" and "SAVE".

A „Save” gombra kattintva menhetjük el a változtatásokat



## Prioritás

A teendőinknek prioritása van, ez azt jelenti, hogy a sorrendje a teendőknek prioritás szerint csökkenő sorrendben látjuk, a kisebb szám nagyobb prioritást jelent.

<input checked="" type="checkbox"/> <b>Second task</b> i also need to do this Priority: 1	 	<input type="checkbox"/> <b>I edited this task's description</b> I really really need to do this Priority: 1	 
<input type="checkbox"/> <b>I edited this task's description</b> I really really need to do this Priority: 2	 	<input type="checkbox"/> <b>Mytask</b> i need to do this Priority: 3	 
<input type="checkbox"/> <b>Mytask</b> i need to do this Priority: 3	 	<input checked="" type="checkbox"/> <b>Second task</b> i also need to do this Priority: 3	 
<input type="checkbox"/> <b>Its third time</b> i havent done this Priority: 3	 	<input type="checkbox"/> <b>Its third time</b> i havent done this Priority: 3	 



E két kép között csak a „Second task” prioritását vettem nagyobbra, és az „I edited this task’s description” teendőét vettem lejjebb

# Megvalósítás

Az alkalmazás MVC (Model-View-Controller), vagy itt MVVM (Model-View-ViewModel) nevű architektúrával van megvalósítva, ennek részletei a következők

## Adatbázis

Az alkalmazás RoomDatabase-t használ, ez egy Android Jetpack könyvtár része, és egy ORM-ként működik az SQLite felett.

Célunk vele, hogy az alkalmazás adatait perzisztensé tegyük, azaz ne tűnjön el az összes adat amikor kilépünk az alkalmazásból

### Task model

Első lépés létrehozni egy modellt, hogy az adatbázisban milyen adatot fogunk tárolni. Ez egy Task entitás lesz, amelyben a @Entity annotációval jelöljük a RoomDB-nek, hogy ő lesz a modellünk

```
6  @Entity(tableName = "tasks")
7  data class Task(
8      @PrimaryKey(autoGenerate = true) val id: Int = 0,
9      val title: String,
10     val description: String,
11     val priority: Int,
12     val isCompleted: Boolean = false
13 )
14
```

## Data Access Object létrehozása (DAO)

DAO-k segítségével határozzuk meg a lekérdezéseket amiket az alkalmazás végrehajt, itt is annotációkkal jelöljük, hogy mit szeretne csinálni az a függvény, itt az alap CRUD műveleteket definiáljuk, csak az adatok lekérdezésénél bővítjük a lekérdezést saját paranccsal , így a teendők prioritás szerinti sorrendben jelennek meg.

```
1 package com.example.taskmanager
2
3 > import ...
4
5
6
7
8
9
10
11 @Dao
12 interface TaskDao {
13     @Insert(onConflict = OnConflictStrategy.REPLACE)
14     fun insert(task: Task): Long
15
16     @Update
17     fun update(task: Task): Int
18
19     @Delete
20     fun delete(task: Task) : Int
21
22     @Query("SELECT * FROM tasks ORDER BY priority ASC")
23     fun getAllTasks(): LiveData<List<Task>>
```



## Adatbázis létrehozása

Ezután a szintén RoomDatabase részeként, a @Database annotációval egy absztrakt osztályt készítünk, amiben definiáljuk hogy az adatbázis milyen DAO szerint működjön.

```
@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDatabase : RoomDatabase() {

    ▶ Kristof
    abstract fun taskDao(): TaskDao

    ▶ Kristof
    companion object {
        @Volatile
        private var INSTANCE: TaskDatabase? = null

        ▶ Kristof
        fun getDatabase(context: Context): TaskDatabase {
            return INSTANCE ?: synchronized(lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    TaskDatabase::class.java,
                    name: "task_database"
                ).build()
                INSTANCE = instance
                instance ^synchronized
            }
        }
    }
}
```

Dependency Injectiont hajtunk rajta végre, majd a companion object segítségével az osztály függvényeit és adattagjait definiáljuk (companion object hasonlít a statichoz, csak az nincs a Kotlinban)

## TaskViewModel

Az az MVVM architektúra következő lépése, ez fog összekapcsolást engedni az alkalmazás MainActivity részével (amelyben definiáljuk az alkalmazás működését), itt hozzárendeljük a TaskRepositoryt, amely engedélyezni fogja az alkalmazásnak, hogy kezelje az adatokat

```
class TaskViewModel(application: Application) : AndroidViewModel(application) {

    private val repository: TaskRepository
    val allTasks: LiveData<List<Task>>

    Kristof
    init {
        val taskDao = TaskDatabase.getDatabase(application).taskDao()
        repository = TaskRepository(taskDao)
        allTasks = repository.allTasks
    }

    Kristof
    fun insert(task: Task) {
        viewModelScope.launch { this: CoroutineScope
            repository.insert(task)
        }
    }

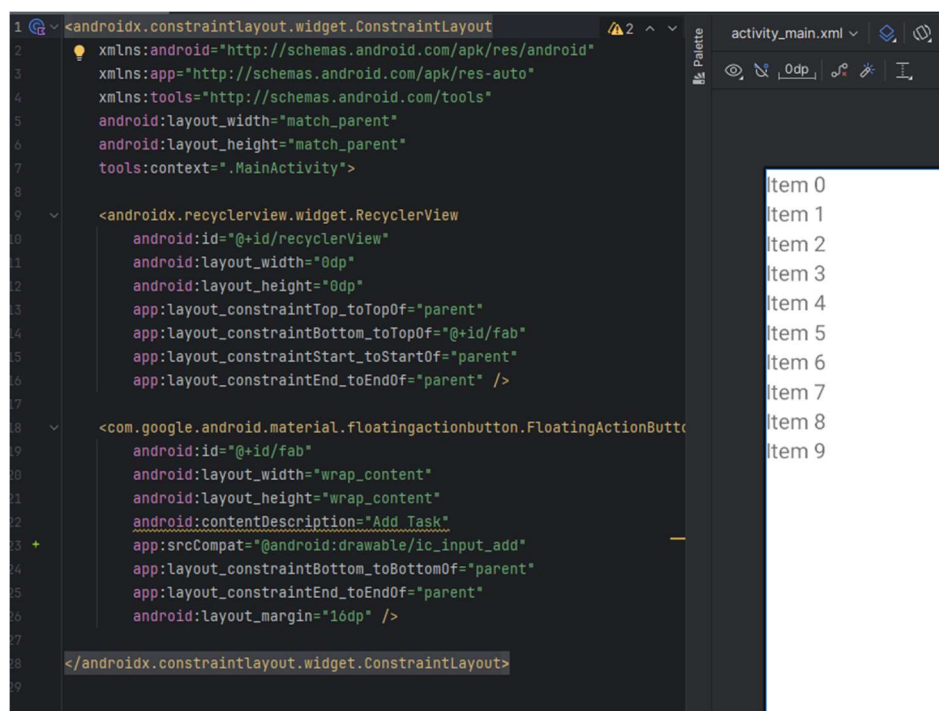
    Kristof
    fun update(task: Task) {
        viewModelScope.launch { this: CoroutineScope
```

## Az alkalmazás függőségei

A build.gradle.kts fájlban a gradle-nek amivel az alkalmazást buildeljük megmondjuk az alkalmazás Android API verziószámát, compiler verziókat, stb. Itt a dependencies blokkba írjuk az alkalmazás függőségeit, ez fogja telepíteni nekünk őket ( mint az npm). Az alkalmazás megjelenéséhez használtak a Material-t, amely segítségével kevesebbet kellett stílusolgatnom az alkalmazás fejlesztése közben.

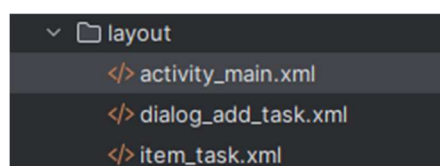
## Layout

Az alkalmazás kinézetét a Layout XML fileokban definiáljuk, a következő példában a főképernyő Layoutját láthatjuk



Itt adjuk hozzá Listás megjelenést, valamint a gombot amivel teendőt tudunk hozzáadni.

A többi komponenst a többi layout fileban definiáltam.



## MainActivity

Miután definiáltuk az alkalmazás kinézetét, és műveleteit, a MainActivity fileban tudjuk összerendelni őket, hogy melyik Layout elemhez melyik művelet tartozzon

Ennek az osztálynak az összes előbbi objektumát példányosítjuk, majd megjelöljük hogy melyik elemhez tartozzon az adott művelet és hogy miként viselkedjen.

Egy példa erre amit a következőben láthatunk, a teendő létrehozása (vagy módosítása) működésének definiálása

```
private fun showTaskDialog(task: Task?) {
    val dialogBinding = layoutInflater.inflate(R.layout.dialog_add_task, root: null)

    val titleInput = dialogBinding.findViewById<EditText>(R.id.etTaskTitle)
    val descriptionInput = dialogBinding.findViewById<EditText>(R.id.etTaskDescription)
    val prioritySpinner = dialogBinding.findViewById<Spinner>(R.id.spinnerPriority)
    val completedCheckbox = dialogBinding.findViewById<CheckBox>(R.id.cbCompleted)

    val priorityList = listOf("High", "Medium", "Low")
    val adapter = ArrayAdapter(
        context: this,
        android.R.layout.simple_spinner_item,
        priorityList
    )
    adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
    prioritySpinner.adapter = adapter

    task?.let { it: Task ->
        titleInput.setText(it.title)
        descriptionInput.setText(it.description)
        prioritySpinner.setSelection(it.priority - 1)
        completedCheckbox.isChecked = it.isCompleted
    }

    MaterialAlertDialogBuilder(context: this)
        .setView(dialogBinding)
        .setTitle(if (task == null) "Add Task" else "Edit Task")
        .setPositiveButton(text: "Save") { _, _ ->
            val title = titleInput.text.toString().trim()
            val description = descriptionInput.text.toString().trim()
            val priority = prioritySpinner.selectedItemPosition + 1
            val isCompleted = completedCheckbox.isChecked

            if (title.isNotEmpty()) {
                val newTask = Task(
                    id = task?.id ?: 0,
                    title = title,
                    description = description,
                    priority = priority,
                    isCompleted = isCompleted
                )
                lifecycleScope.launch(Dispatchers.IO) { this: CoroutineScope ->
                    if (task == null) taskDao.insert(newTask) else taskDao.update(newTask)
                }
            } else {
                Toast.makeText(context: this, text: "Task title is required", Toast.LENGTH_SHORT).show()
            }
        }
        .setNegativeButton(text: "Cancel", listener: null)
        .show()
}
```

Ezzel a kódrészlettel eldöntjük, hogy melyik gombra mentünk (plusz vagy ceruza) és az alapján fogja címezni a dialógus dobozt, majd a Layoutban definiált szövegdobozokból az adatot kisedjük, és a „Save”-re kattintva az adatbázisba feltöltjük, vagy módosítjuk az adatot.