Task HD – Spike: [Notes App]

Goals: Android App Development

The goal is to create a Notes App where you can add notes, edit them and delete them. The notes will be stored in a database and will be loaded from them every time the app is launched. Moreover, there is a search bar as well where you can search for specific keywords in your notes to find the required note. The color of each note is assigned randomly so they're always different from each other. The notes can be deleted using context menus when it is being clicked long enough. If there is a shot click the notes page will open up and you can update it.

The following list outlines the goal broken down into more specific knowledge gaps involved in the goal.

- *Creating Clickable items in recycler view*
- *Using recycler view and adapter*
- *Using context menus*
- *Transferring data using intents*
- *Storing & fetching data from database*
- *Adding new notes*
- *Updating notes*
- *Deleting notes*
- *Creating Search bar*
- *Debug and testing*
- *Creating Clickable button*

1. *Logs*
2. *Unit testing*
3. *Integration testing*

Tools and Resources Used

This section lists related software, tools, libraries, API's, and other resources used for this knowledge gap.

- Android Studio
- Android Documentation
- You tube
- Canvas notes

Knowledge Gaps and Solutions

This section presents the listed knowledge gaps and their solutions with supporting images, screenshots and captions where appropriate/required.

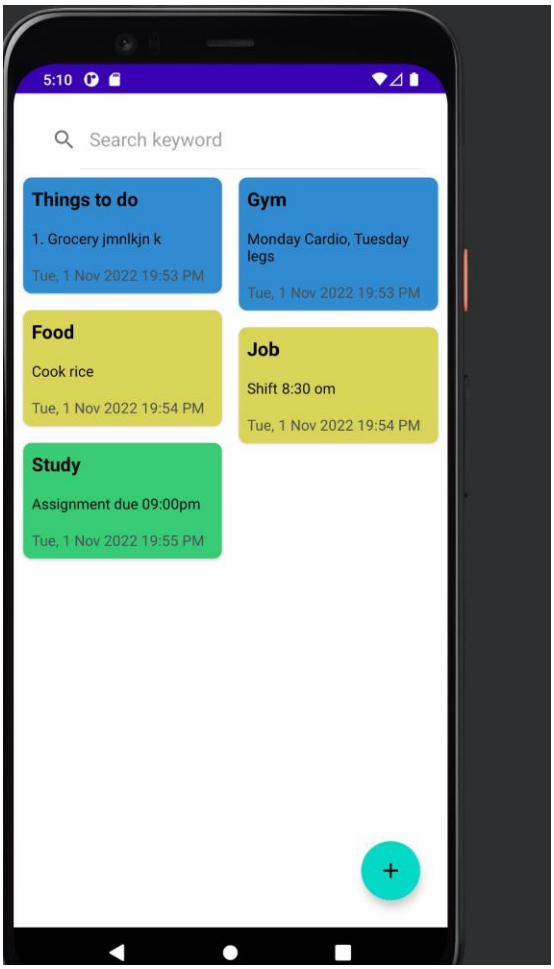
Gap 1 (Creating Clickable items in list view) :

A recycler view is created in which each item is clickable and a pop shows up when an item is clicked.

```kotlin
    private fun initUI(){
        binding.recyclerView.setHasFixedSize(true)
        binding.recyclerView.layoutManager = StaggeredGridLayoutManager( spanCount: 2,LinearLayout.VERT
        adapter = NotesAdapter( context: this,  listner: this)
        binding.recyclerView.adapter = adapter

        val getContent = registerForActivityResult(ActivityResultContracts.StartActivityForResult()){
            if (result.resultCode == Activity.RESULT_OK){
                val note = result.data?.getSerializableExtra( name: "note") as? Note
                if(note != null){
                    viewModel.insertNote(note)
                }
            }
        }
        binding.floatingActionButton.setOnClickListener{  it: View!
            val  intent = Intent( packageContext: this, AddNote::class.java)
            getContent.launch(intent)
        }

        binding.searchView.setOnQueryTextListener(object : SearchView.OnQueryTextListener{
            override fun onQueryTextSubmit(p0: String?): Boolean {
                return false
            }

            override fun onQueryTextChange(newText: String?): Boolean {
                if (newText!= null){
                    adapter.filterList((newText))
                }
```

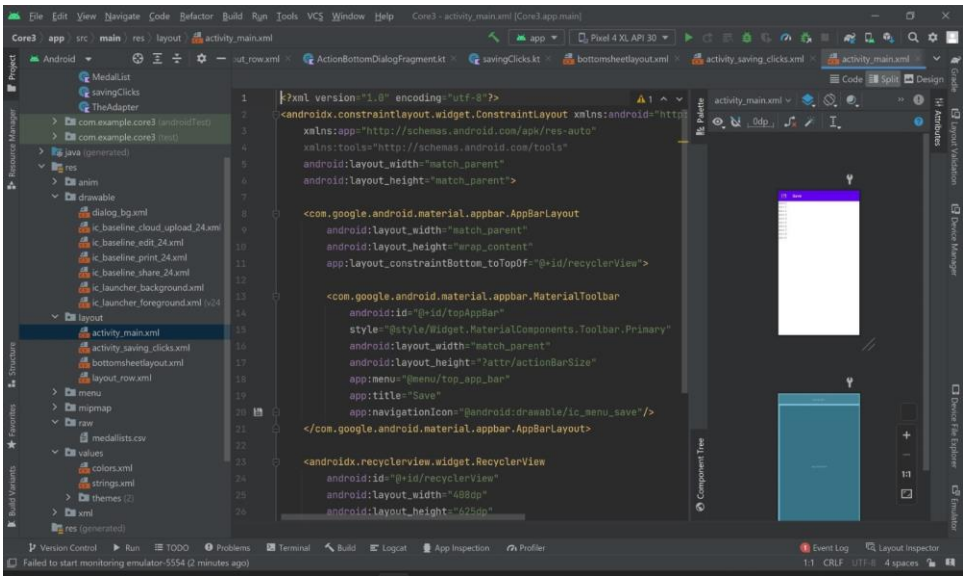
```kotlin
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): NoteViewH
        return NoteViewHolder(
            LayoutInflater.from(context).inflate(R.layout.list_item, parent,  attachToRoot: false)
        )
    }

    override fun onBindViewHolder(holder: NoteViewHolder, position: Int){
        val  currentData = NotesList[position]
        holder.title.text = currentData.title
        holder.title.isSelected = true
        holder.Note_tv.text = currentData.note
        holder.date.text = currentData.date
        holder.date.isSelected = true

        holder.notes_layout.setCardBackgroundColor(holder.itemView.resources.getColor(randomColor(),
        holder.notes_layout.setOnClickListener{  it: View!
            listner.onItemClicked(NotesList[holder.adapterPosition])
        }
        holder.notes_layout.setOnLongClickListener{  it: View!
            listner.onLongItemClicked(NotesList[holder.adapterPosition],holder.notes_layout)
            true  ^setOnLongClickListener
        }
    }

    override fun getItemCount(): Int {
        return NotesList.size
    }
```

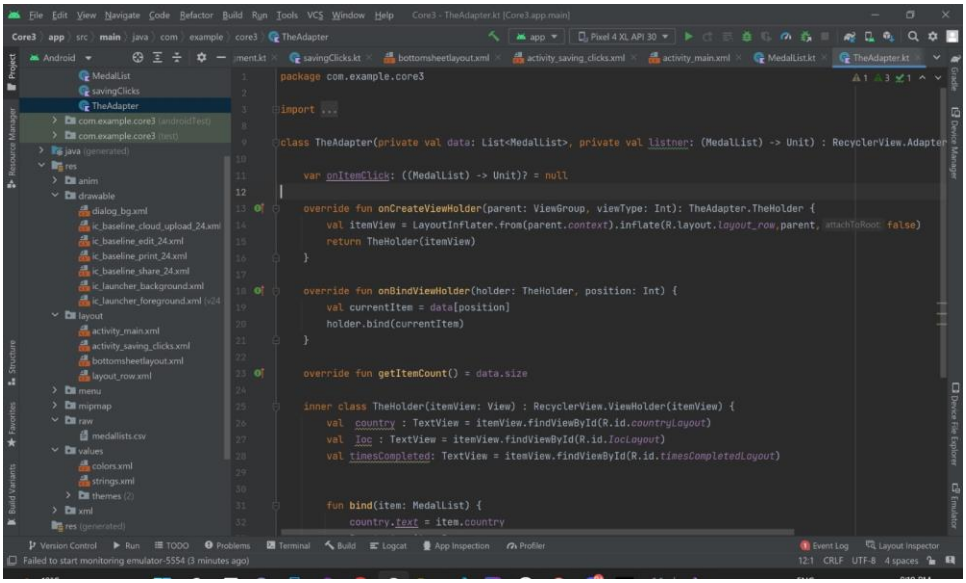
It will always be more efficient to use more specific change events if you can. Rely on `notifyDataSetChanged` as a last resort.



The above screenshots show the adapter class and the main activity code for creation of the recycler view. In the adapter class an updateList method is added which can be used to detect which item in the recycler view is selected

Gap 2(Using Recycler View and Adapter):





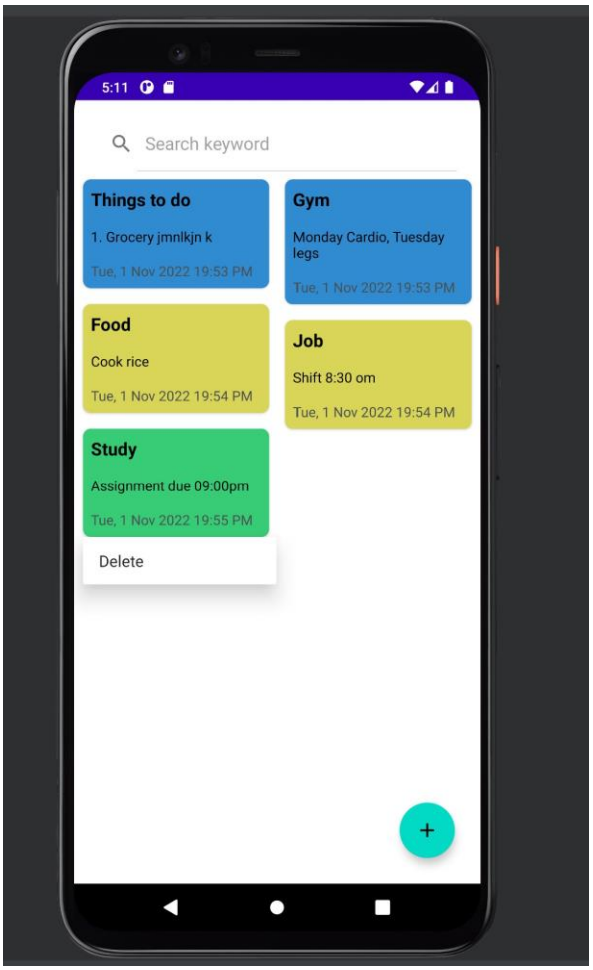
Recycler View has been used to display the item of the file in a list. The above screenshots show the design file of recycler view and the adapter class code for that.

Gap 3 (Using context menus):

Context menus have been used with the onLongItemClickListner() method. When a note is clicked for long time a pop up delete option will pop up. Pressing the delete button will result in the specific not being deleted from the database and this will refresh the activity as well so it won't appear on the screen as well.

```kotlin
private fun popUpDisplay(cardView: CardView){
    val popup = PopupMenu( context: this, cardView)
    popup.setOnMenuItemClickListener(this@MainActivity)
    popup.inflate(R.menu.pop_up_menu)
    popup.show()
}
```

```kotlin
override fun onMenuItemClick(item: MenuItem?): Boolean {
    if (item?.itemId == R.id.delete_note){
        viewModel.deleteNote(selectedNote)
        return true
    }
    return false
}
```



Gap 4 (Transferring data using intents):

Intents were used to switch between screens when a note is clicked. The .putExtra function was used to tranfer values of heading and note description. Moreover, in the detail activity edit texts are used to change values in the array which also update it on the main activity.

```
override fun onItemClicked(note: Note) {
    val intent = Intent( packageContext: this@MainActivity, AddNote::class.java)
    intent.putExtra( name: "current_note", note)
    updateNote.launch(intent)
}
```

```
private val updateNote = registerForActivityResult(ActivityResultContracts.StartActivityForResult()){result ->
    if (result.resultCode == Activity.RESULT_OK){
        val  note = result.data?.getSerializableExtra( name: "note") as? Note
        if (note != null){
            viewModel.updateNote(note)
        }
    }
}
```

Gap 5 (Storing and fetching data from database):

The following code was used to store and fetch data from the database so every time the app will show it when it is loaded and it won't get deleted when we turn off the app. SQL Lite has been used for this purpose. An Interface is created for this purpose which contains the SQL queries. Database is accessed through an instance and the view model is used to view the fetched data from the database.

```
NotesRepository.kt  MainActivity.kt  constants.kt  Note.kt  NotesViewModel.kt  AddNote.kt  NoteDao.kt  NoteDatabase
package com.example.notesapp.Models

import ...

@Entity(tableName = "notes_table")
data class Note(
    @PrimaryKey(autoGenerate = true) val id: Int?,
    @ColumnInfo(name = "title") val title: String?,
    @ColumnInfo(name = "note") val note: String?,
    @ColumnInfo(name = "date") val date: String?
): java.io.Serializable
```

```kotlin
package com.example.notesapp.Database

import ...

@Dao
interface NoteDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(note: Note)

    @Delete
    suspend fun delete(note: Note)

    @Query("UPDATE notes_table Set title = :title, note = :note WHERE id = :id")
    suspend fun update(id: Int?, title: String?, note: String?)

    @Query("Select * from notes_table order by id ASC")
    fun  getAllNotes(): LiveData<List<Note>>
}
```

```kotlin
package com.example.notesapp.Database

import ...

@Database(entities = arrayOf(Note::class), version = 1, exportSchema = false)
abstract class NoteDatabase: RoomDatabase() {

    abstract fun getNoteDao(): NoteDao
    companion object{
        @Volatile
        private var INSTANCE: NoteDatabase? = null

        fun getDatabase(context: Context): NoteDatabase{
            return INSTANCE ?: synchronized( lock: this){
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    NoteDatabase::class.java,
                    DATABASE_Name
                ).build()

                INSTANCE = instance
                instance  ^synchronized
            }
        }
    }
}
```

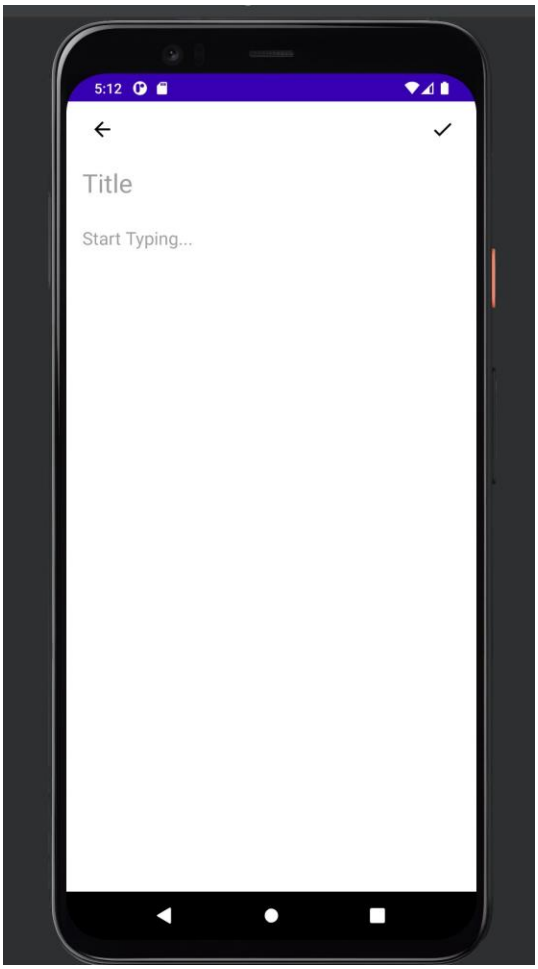
```kotlin
package com.example.notesapp.Database

import ...

class NotesRepository(private val noteDao: NoteDao){
    val allNotes: LiveData<List<Note>> = noteDao.getAllNotes()

    suspend fun insert(note: Note){
        noteDao.insert(note)
    }

    suspend fun update(note: Note){
        noteDao.update(note.id, note.title, note.note)
    }

    suspend fun delete(note: Note){
        noteDao.delete(note)
    }
}
```

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

    private val repository: NotesRepository
    val allnotes: LiveData<List<Note>>
    init {
        val dao = NoteDatabase.getDatabase(application).getNoteDao()
        repository = NotesRepository(dao)
        allnotes = repository.allNotes
    }

    fun deleteNote(note: Note) = viewModelScope.launch(Dispatchers.IO){
        repository.delete(note)
    }

    fun insertNote(note: Note) = viewModelScope.launch(Dispatchers.IO){
        repository.insert(note)
    }

    fun updateNote(note: Note) = viewModelScope.launch(Dispatchers.IO){
        repository.update(note)
    }
}
```

Gap 6 (Adding new notes):

The following code has been used to add a new note to the database. This code also includes the part which shows how data is being transferred using intents.

```
package com.example.notesapp

import ...

class AddNote : AppCompatActivity() {

    private lateinit var binding: ActivityAddNoteBinding
    private lateinit var note: Note
    private lateinit var old_note: Note
    var isUpdate = false
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityAddNoteBinding.inflate(layoutInflater)
        setContentView(binding.root)

        try {
            old_note = intent.getSerializableExtra("current_note") as Note
            binding.txtTitle.setText(old_note.title)
            binding.txtNote.setText(old_note.note)
            isUpdate = true

        }catch (e: Exception){
            e.printStackTrace()
        }
        binding.imgCheck.setOnClickListener{
            val title = binding.txtTitle.text.toString()
            val note_desc = binding.txtNote.text.toString()
```



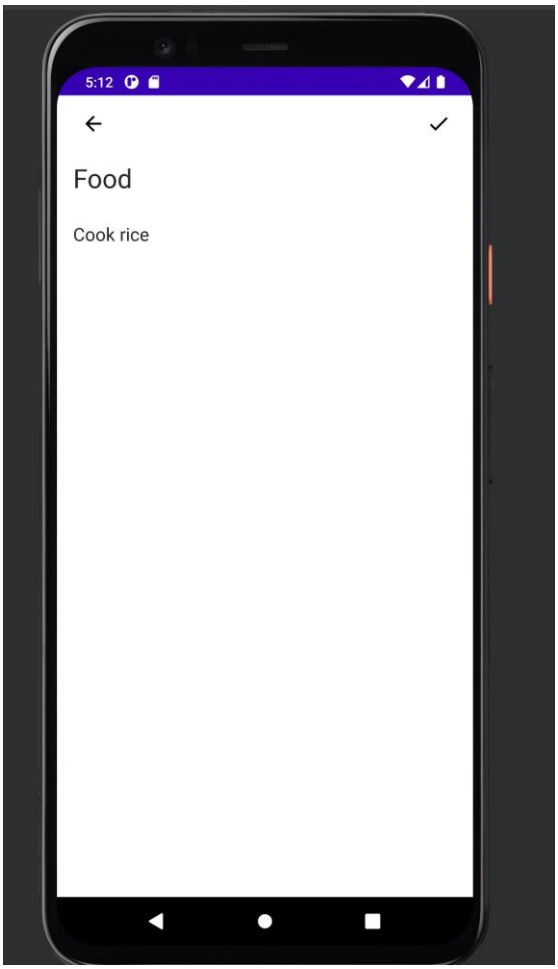
Gap 7 (Updating notes):

The following code was used to update an existing note. At first the data already stored in the node is fetched from the data base to the editing page so the user can see what has been previously added. This can then be updated and will be shown on the notes page.

```
private val updateNote = registerForActivityResult(ActivityResultContracts.StartActivityForResult()){result ->
    if (result.resultCode == Activity.RESULT_OK){
        val  note = result.data?.getSerializableExtra( name: "note") as? Note
        if (note != null){
            viewModel.updateNote(note)
        }
    }
}
```

```
suspend fun update(note: Note){
    noteDao.update(note.id, note.title, note.note)
}
```

```
fun updateNote(note: Note) = viewModelScope.launch(Dispatchers.IO){  this: CoroutineScope
    repository.update(note)
}
```



Gap 7 (Deleting notes):

The following code was used to delete the note from the database. The note clicked is noted and the id of that note is used to delete the note from the database.

```
fun deleteNote(note: Note) = viewModelScope.launch(Dispatchers.IO){ this: CoroutineScope
    repository.delete(note)
}
```

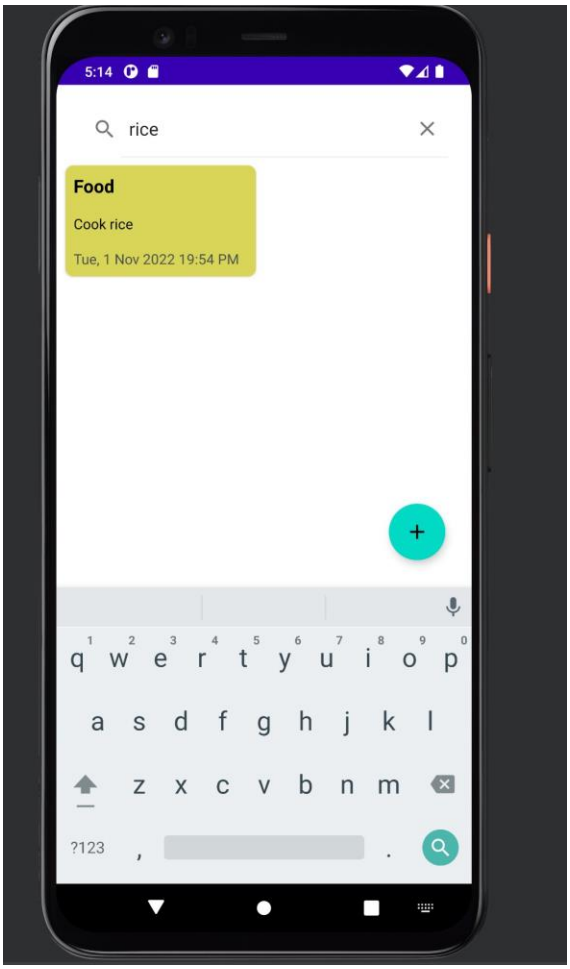
```
private fun popUpDisplay(cardView: CardView){
    val popup = PopupMenu( context: this, cardView)
    popup.setOnMenuItemClickListener(this@MainActivity)
    popup.inflate(R.menu.pop_up_menu)
    popup.show()
}

override fun onMenuItemClick(item: MenuItem?): Boolean {
    if (item?.itemId == R.id.delete_note){
        viewModel.deleteNote(selectedNote)
        return true
    }
    return false
}
```

Gap 8 (Creating Seach bar):

The following code has been used to filter the notes according to the keywords typed. It clears the notes list and then search for the item specified in search bar. If the keywords is found it add that nots in the notes list which is then displayed. This function is triggered everytime there is a data set change in the notes list.

```
fun filterList(search: String){
    NotesList.clear()
    for (item in fullList){
        if (item.title?.lowercase()?.contains(search.lowercase()) == true || item.note?.lowercase()?.contains(search.lowercase()) == true){
            NotesList.add(item)
        }
    }
    notifyDataSetChanged()
```

```
binding.searchView.setOnQueryTextListener(object : SearchView.OnQueryTextListener{
    override fun onQueryTextSubmit(p0: String?): Boolean {
        return false
    }

    override fun onQueryTextChange(newText: String?): Boolean {
        if (newText!= null){
            adapter.filterList((newText))
        }
        return true
    }
})
```



Gap 9 (Debug and Testing):

- Logs were used to test that data is being read through the file or not.
- Breakpoints were used to stop the program to test it. This helps to check the variable internally when the score button is pressed.

- Unit testing was done to check each method is working or not working
- Integration testing was done to check how activities interact with each other

Open Issues and Recommendations

This section outlines any open issues, risks, and/or bugs, and highlights potential approaches for trying to address them in the future.

What is the right method to implement onClick Listener?

Normally, there are two principal ways of carrying out onClickListener event on Button, and the method can be applied to other event handlers. We can create and define an event listener for each button

inside the onCreate() function like so.

We're unsure about how many buttons do we have so another way to implement the event listner.

```
popup.setOnMenuItemClickListener(this@MainActivity)
```

Architectural Design Diagram: