



Room part 2

Week 9

Advanced Native Mobile Programming

Teknik Informatika - Universitas Surabaya

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting a hierarchical or multi-layered structure. The lines are thin and gray, connecting the nodes in a non-linear fashion.

0. **The Concept**

Database Migration

- Database migration is the process of migrating data from one or more source databases to one or more target databases by using a database migration service.
- The **Room** persistence library supports incremental migrations with the **Migration class** to address this need

How Migration Works

- ◎ Version 1
User { id:Int, name:String, }
- ◎ Version 2
User { id: Int, name:String, email:String }
- ◎ Version 3
User { id: int, name:String, email:String, password:String }
- ◎ At first installation app will execute migration incrementally starting from 1 to 2 and finally 2 to 3.

Migration Class

Each Migration subclass defines a migration path between a **startVersion** and an **endVersion** by overriding the `Migration.migrate()` method.

```
val MIGRATION_1_2 = object : Migration(1, 2) {  
    override fun migrate(database: SupportSQLiteDatabase) {  
        database.execSQL  
            ("ALTER TABLE user ADD COLUMN email TEXT")  
    }  
}
```

```
val MIGRATION_2_3 = object : Migration(2, 3) {  
    override fun migrate(database: SupportSQLiteDatabase) {  
        database.execSQL  
            ("ALTER TABLE user ADD COLUMN password TEXT")  
    }  
}
```

Migration Class

Room can handle more than one version increment: we can define a migration that goes from version 1 to 3 in a single step, making the migration process faster.

```
val MIGRATION_1_3 = object : Migration(1, 3) {  
    override fun migrate(database: SupportSQLiteDatabase) {  
        database.execSQL  
            ("ALTER TABLE user ADD COLUMN email TEXT")  
        database.execSQL  
            ("ALTER TABLE user ADD COLUMN password TEXT")  
    }  
}
```

Add the migration to the Room database builder

```
Room.databaseBuilder(applicationContext, MyDb::class.java,  
    "database-name")  
    .addMigrations(MIGRATION_1_2, MIGRATION_2_3, MIGRATION_1_3).build()
```

Destructive Migration

- ◎ In some cases, migrations is not needed and its okay to clear all record from the database during the upgrade version, then you may call **fallbackToDestructiveMigration** in the database builder:

```
val db= Room.databaseBuilder(context.getApplicationContext(),  
                             MyDb.class, "database-name")  
                             .fallbackToDestructiveMigration()  
                             .build();
```

Auto Migration

- Starting from Room 2.4.0-alpha01, we can also use auto migration when the changes are simple (e.g. adding new column to an existing table)
- Read more:
<https://medium.com/androiddevelopers/room-auto-migrations-d5370b0ca6eb>

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels of connectivity or importance. The lines are thin and gray, creating a mesh-like structure.

1. Utility

Utility Package

- ◎ The building database methods have been placed in a different location. It'd be better if we centralized that method within a single function in the utility file.
- ◎ Create new package > util
- ◎ Create new file, name it Util.kt and put it under the package



Util

buildDB

This is the same function that use to gain access to DB resources. It requires context and returns the TodoDatabase DB object.

```
val DB_NAME = "newtododb"
```

```
fun buildDb(context: Context): TodoDatabase {  
    val db = Room.databaseBuilder(context, TodoDatabase::class.java, DB_NAME)  
        .build()  
    return db  
}
```

Call buildDb in ViewModel

Next, check all ViewModel (DetailTodoViewModel & ListTodoViewModel) class and replace the previous hardcoded build database function with the buildDb method from Util

```
val db = buildDb(getApplication())
```

Create Soft Back Button

Next, setup the **main activity** to show soft back button when needed.

```
private lateinit var navController: NavController

override fun onCreate(savedInstanceState: Bundle?) {
    . . .
    navController = (supportFragmentManager.findFragmentById(R.id.fragmentHost) as
    NavHostFragment).navController
    NavigationUI.setupActionBarWithNavController(this, navController)
}

override fun onSupportNavigateUp(): Boolean {
    return NavigationUI.navigateUp(navController, null)
}
```

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels or types of nodes. The lines are thin and gray, connecting the nodes in a non-uniform, organic pattern.

2. **The Layout**

Add Priority Radio Button

Open the Create Todo Fragment layout:

1. Add Radio Group
2. Add three Radio Button into Radio Group
3. Name the RadioGroup id as “radioGroupPriority”
4. Name each radio button id as “radioHigh”, “radioMedium”, and “radioLow” respectively from top to bottom



Priority

☒ High Priority

☐ Medium Priority

☐ Low Priority



```
▼ radioGroupPriority
  radioHigh "@st...
  radioMedium "...
  radioLow "@str...
```

Set Tag Value

Each radio button can have a tag value. It's handy to set a value for each radio button

- ☒ radioHigh, set tag to 3
- ☐ radioMedium, set tag to 2
- ☐ radioLow, set tag to 1

This value is used to set todo priority. A high number means high priority.

tag	3	
-----	---	--

Default Selected Radio Button

Click on Radio button High Priority and set checked to true. Therefore this radio button is selected as default.

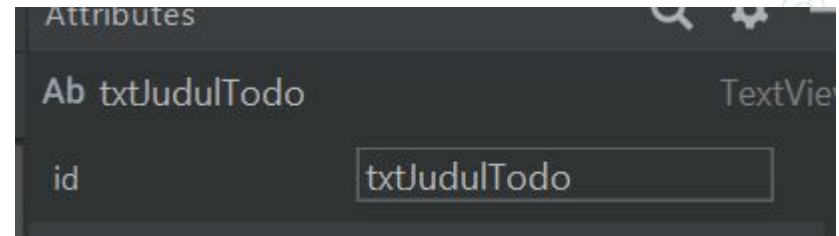
Priority

☒ High Priority

checked	<input checked="" type="checkbox"/> true
---------	--

Set Id for Todo Header Titler

Click on new todo Header



Set ID

By setting the id for this textview, it can be changed programmatically. It'll be coming in handy later.

Create New Fragment

- ◎ Next, create a new Edit Fragment to edit the todo data.
- ◎ Name it as “EditTodoFragment”
- ◎ Since this fragment will reuse the same layout from Create Todo Fragment, you may **delete the fragment_edit_todo.xml**

CleanUp Codes

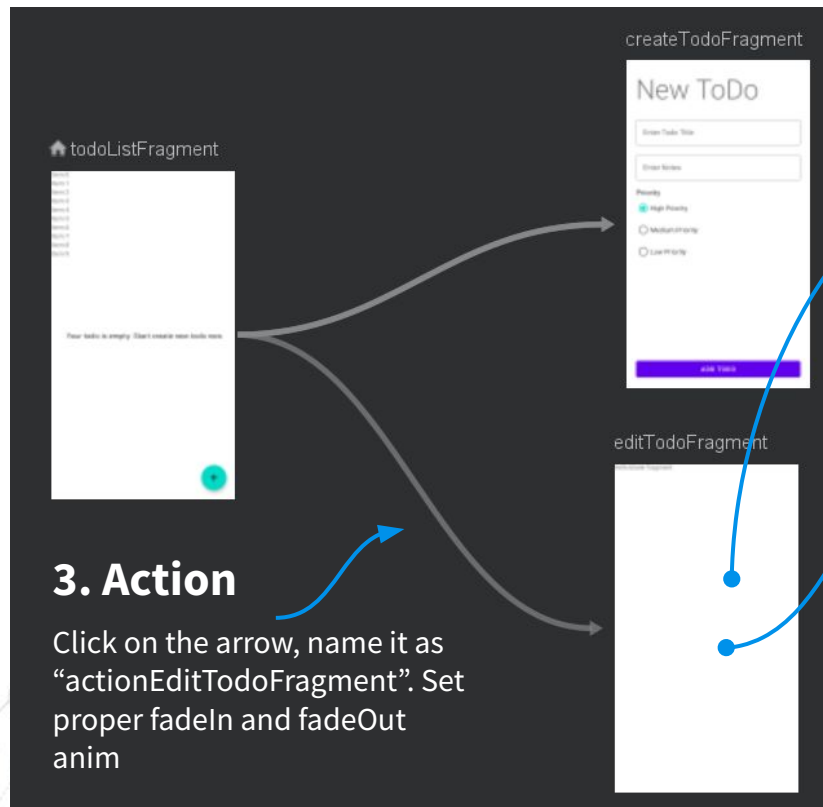
- ◎ As usual, delete everything in EditTextoFragment except onCreateView
- ◎ Override the onCreateView

```
override fun onCreateView(view: View, savedInstanceState: Bundle?) {  
    super.onCreateView(view, savedInstanceState)  
}
```

The Navigation Graph

- ◎ This new fragment is a new destination reachable from the `TodoListFragment`
- ◎ User may click on pencil edit button to edit particular todo
- ◎ Open the `navigation_main.xml`

The Navigation Graph



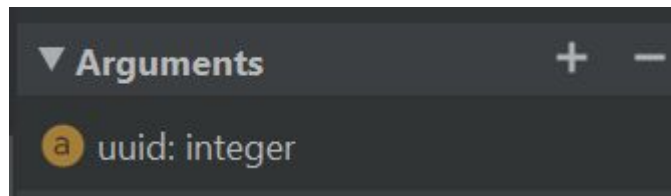
1. New destination

Add the editTodoFragment, and set action (connection) from todoListFragment.

2. Add Argument

Click the editTodoFragment, and then add "uuid" (integer) argument.

This argument used to load particular todo data in Edit Todo Fragment





Call The Navigation

- ◎ Don't forget to rebuild the project, to apply the changes of Navigation Graph
- ◎ Next, call the navigation action from the adapter
- ◎ Open the TodoListAdapter

Call The Navigation

```
holder.view.imgEdit.setOnClickListener {  
    val action =  
        TodoListFragmentDirections.actionEditTodoFragment(todoList[position].uuid)  
  
    Navigation.findNavController(it).navigate(action)  
}
```

Edit Listener

This code is used to setup the listener whenever user clicks on pencil edit button. It navigates to Edit Todo Fragment

Notice that it also passes the argument of selected uuid

```
holder.view.checkTask.setOnCheckedChangeListener { compoundButton, isChecked ->  
    if(isChecked == true) {  
        adapterOnClick(todoList[position])  
    }  
}
```

Check Task

Make little adjustments for check task listener. Now if the task is checked, then it should be removed from database

The ViewModel

Remember, there is no LiveData variable in the DetailTodoViewModel. Next step is to create new one (it will be used by Edit Todo Fragment).

- ⦿ Open the DetailTodoViewModel.kt
- ⦿ Add the Live Data variable

```
val todoLD = MutableLiveData<Todo>()
```

The ViewModel

🕒 Add fetch function with uuid as parameter

```
fun fetch(uuid:Int) {  
    launch {  
        val db = buildDb(getApplication())  
        todoLD.value = db.todoDao().selectTodo(uuid)  
    }  
}
```

Select Todo

This fetch function use selectTodo from DAO. This will load a single Todo and returns it.

The ViewModel

The Edit Todo fragment requires the same ViewModel that being used by the CreateTodoFragment

- ⊙ Open the **EditTodoFragment.kt**

- ⊙ Create the viewmodel variable

```
private lateinit var viewModel: DetailTodoViewModel
```

- ⊙ Intialize the viewmodel in **onViewCreated**

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
    super.onViewCreated(view, savedInstanceState)  
    viewModel = ViewModelProvider(this).get(DetailTodoViewModel::class.java)  
}
```

Rename the Title and Button Add

Remember, this fragment uses the same layout from Create Todo. Therefore the title needs to be adjusted properly. Add following codes inside **onViewCreated**:

```
txtJudulTodo.text = "Edit Todo"  
btnAdd.text = "Save Changes"
```

Load the Data

```
viewModel = ViewModelProvider(this).get(DetailTodoViewModel::class.java)
```

```
val uuid = EditTodoFragmentArgs.fromBundle(requireArguments()).uuid
```

```
viewModel.fetch(uuid)
```

```
observeViewModel()
```

1. Read UUID

First, it read argument send from the Adapter

2. Fetch

Second, it called fetch from live model with uuid supplied as parameter

3. Observe

Finally, observe the live data (todo)

Observe Method

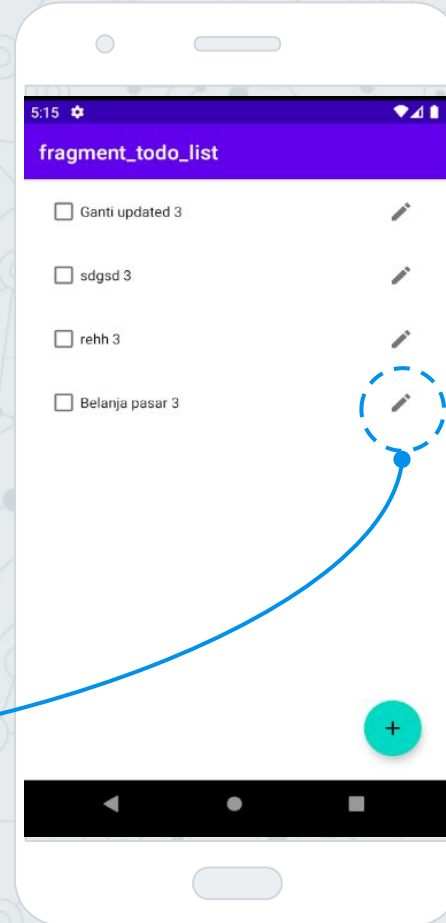
```
fun observeViewModel() {  
    viewModel.todoLD.observe(viewLifecycleOwner, Observer {  
        txtTitle.setText(it.title)  
        txtNotes.setText(it.notes)  
    })  
}
```

Populate

Show the loaded Todo into edit text title and notes

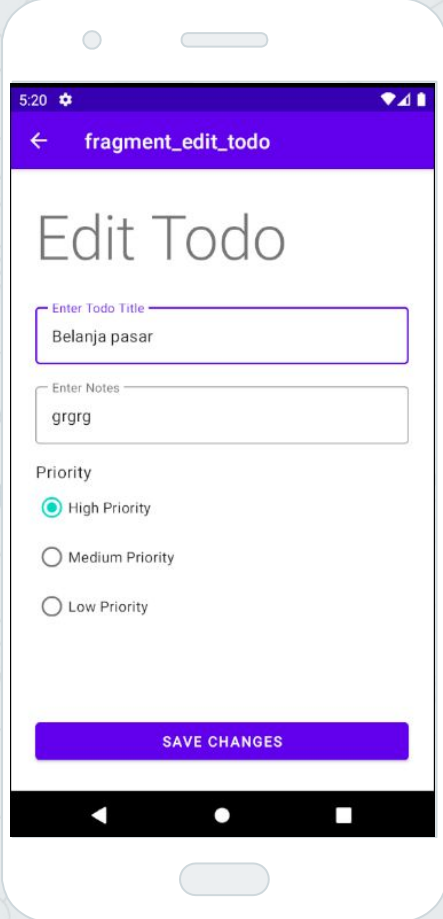
Result

Click on Edit Button



Result

Data loaded in Edit Fragment



The image shows a smartphone screen displaying the 'Edit Todo' interface. The status bar at the top shows the time 5:20, a star icon, and signal/battery indicators. The app bar is purple with a back arrow and the text 'fragment_edit_todo'. The main title is 'Edit Todo'. There are two text input fields: the first is labeled 'Enter Todo Title' and contains the text 'Belanja pasar'; the second is labeled 'Enter Notes' and contains the text 'grgrg'. Below these is a 'Priority' section with three radio button options: 'High Priority' (which is selected), 'Medium Priority', and 'Low Priority'. At the bottom is a purple button labeled 'SAVE CHANGES'.

5:20 ★

← fragment_edit_todo

Edit Todo

Enter Todo Title

Belanja pasar

Enter Notes

grgrg

Priority

☒ High Priority

☐ Medium Priority

☐ Low Priority

SAVE CHANGES

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels or types of nodes. The lines are thin and gray, connecting the nodes in a non-linear fashion.

3. **The Model**

Priority Field

The todo model must be altered to provide a priority value that indicates how important the todo is:

- ⦿ Open the model.kt
- ⦿ Add the priority (int), see next slide

Priority Field

@Entity

```
data class Todo(  
    @ColumnInfo(name="title")  
    var title:String,  
    @ColumnInfo(name="notes")  
    var notes:String,  
    @ColumnInfo(name="priority")  
    var priority:Int  
    ) {  
    @PrimaryKey(autoGenerate = true)  
    var uuid:Int =0  
}
```

Update the DAO

Users may edit the todo after it is created. Therefore inside the DAO, add the update query for todo data.

```
@Query("UPDATE todo SET title=:title, notes=:notes, priority=:priority  
      WHERE uuid = :id")  
suspend fun update(title:String, notes:String, priority:Int, id:Int)
```

priority

Notice that priority field have been added here.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines, with some nodes highlighted in blue.

4.

Database Migration

Database Migration

- ◎ Because there is changes on the Todo Table (we add priority field), therefore it needs to define migration policy. So you need to do three things:
 1. Define Migration Policy (method)
 2. Increase Database Version by 1
 3. Add the migration method into the database builder function

Define Migration Policy (method)

Open the util.kt. Add this migration method:

```
val MIGRATION_1_2 = object : Migration(1, 2) {  
    override fun migrate(database: SupportSQLiteDatabase) {  
        database.execSQL(  
            "ALTER TABLE todo ADD COLUMN priority INTEGER DEFAULT 3 not null")  
        }  
    }  
}
```

MIGRATION_1_2

It means this migration is used for upgrading the db from version 1 to 2

ALTER QUERY

New priority column added. Integer type. And because its must have value (see the model), we add the “DEFAULT 3 not null”. Default 3 means that this priority will be set to 3 for the old data.

Increase Database Version by 1

Open the TodoDatabase.kt:

```
@Database(entities = arrayOf(Todo::class), version = 2)
abstract class TodoDatabase:RoomDatabase() {
    . . .
    companion object {
        . . .
    }
}
```

Version 2

Increase database version by 1, every time you change the data scheme

Add the migration method into the database builder function

Open the TodoDatabase.kt:

```
companion object {  
    . . .  
    private fun buildDatabase(context:Context) = Room.databaseBuilder(  
        context.applicationContext,  
        TodoDatabase::class.java, "newtododb")  
        .addMigrations(MIGRATION_1_2)  
        .build()  
}
```

Add Migration Policy

Add all migration policies into the builder (separate with comma).

Add the migration method into the database builder function

Open the Util.kt:

```
fun buildDb(context: Context): TodoDatabase {  
    val db = Room.databaseBuilder(context,  
        TodoDatabase::class.java, DB_NAME)  
        .addMigrations(MIGRATION_1_2)  
        .build()  
  
    return db  
}
```

Add Migration Policy

Add all migration policies into the builder (separate with comma).

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines, with some nodes highlighted in blue.

5.

Fix The Create & Edit Todo

Fix the Create Todo

- 🕒 Open the CreateTodoFragment.kt
- 🕒 Look for the error. It should be inside the btnAdd listener. Because the Todo class now requires the “**priority**” field

Fix the Create Todo

```
btnAdd.setOnClickListener {  
    var radio =  
        view.findViewById<RadioButton>(radioGroupPriority.checkedRadioButtonId)  
  
    var todo = Todo(txtTitle.text.toString(),  
                    txtNotes.text.toString(), radio.tag.toString().toInt())  
  
    val list = listOf(todo)  
    . . .
```

Get The Radio

First get access to Radio Group priority by
findViewById function

Next get access to the selected radio
button (high, medium, or low)

Retrieve Tag value

Radio object now contain the selected
radio button. Use the tag value to retrieve
the priority number (3, 2, or 1)

Then store it inside the Todo object

Fix The Edit Todo

Open the EditTodoFragment. Look for the ObserveViewModel method

```
fun observeViewModel() {  
    viewModel.todoLD.observe(viewLifecycleOwner, Observer {  
        . . .  
        when (it.priority) {  
            1 -> radioLow.isChecked = true  
            2 -> radioMedium.isChecked = true  
            else -> radioHigh.isChecked = true  
        }  
    })  
}
```

Populate The Radio

Based on priority field, select the appropriate Radio button

Save Changes

Open the DetailTodoViewModel.kt add following method:

```
fun update(title:String, notes:String, priority:Int, uuid:Int) {  
    launch {  
        val db = buildDB(getApplication())  
        db.todoDao().update(title, notes, priority, uuid)  
    }  
}
```

Save Todo Changes

This function is used to update current todo based on its uuid

Save Changes

Go back to EditTodoFragment. Add btnCreateTodo listener to execute update query on Todo:

```
btnAdd.setOnClickListener {  
    val radio =  
        view.findViewById<RadioButton>(radioGroupPriority.checkedRadioButtonId)  
    viewModel.update(txtTitle.text.toString(), txtNotes.text.toString(),  
        radio.tag.toString().toInt(), uuid)  
    Toast.makeText(view.context, "Todo updated", Toast.LENGTH_SHORT).show()  
    Navigation.findNavController(it).popBackStack()  
}
```


A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines, with some nodes highlighted in blue.

6.

Fix The Priority Order in TodoList

Fix The Priority Order in TodoList

- ◎ High priority task should be displayed on top of list and lowest priority should be placed on the bottom
- ◎ To accomplish this, simply adjust the select query with “ORDER” command
- ◎ Open the **todo DAO**, add the ORDER command

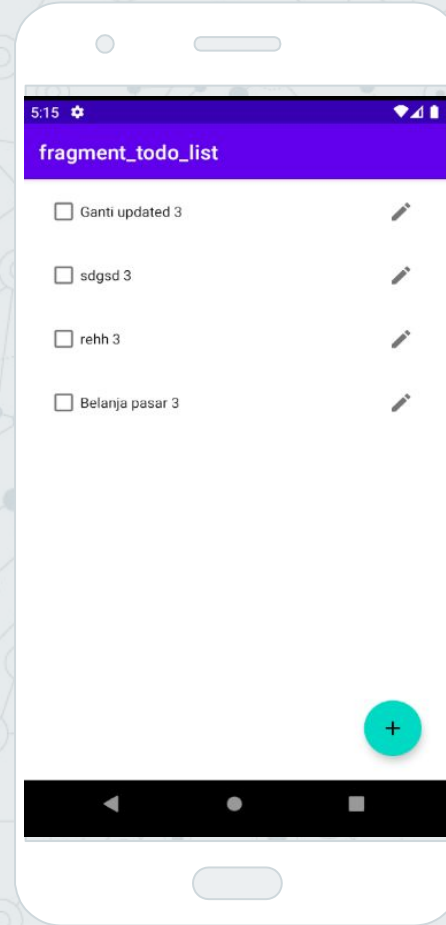
```
@Query("SELECT * FROM todo ORDER BY priority DESC")  
suspend fun selectAllTodo(): List<Todo>
```

Result

Try to create multiple todo with different priority

Try to edit some todos

Try to delete the todos



A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. The nodes are represented by small circles, some of which are larger and have concentric circles, suggesting different levels of connectivity or importance. The lines are thin and gray, creating a mesh-like structure.

7. **Homework**

Homework

- ◎ Alter the Todo table by adding a new field:
 - is_done
 - INTEGER
 - Default value is 0 and not null
- ◎ Every time user **checks the task**, update the todo "is_done" field from 0 to 1. Do not delete the todo. Moreover, only **is_done = 0** that **displayed on the list**.

INTEGER

- ◎ Why this is_done field use INTEGER instead of BOOLEAN?
- ◎ Answer by adding comments on the line where you altered the table, something like this:
`[your modification here] // because ...`

Submission

- ◎ Do the commit and push on the github
- ◎ Set your github project url to **public**
- ◎ Submit the github project url to **ULS**
- ◎ **Due date is next week**



Thanks!

Any questions?

You can reach me at:
andre@staff.ubaya.ac.id