GRANDE ÉCOLE D'INGÉNIEURS

Project Android - Directed Practicals - Part 2

Richard Woodward

September 2024

Contents

1 Application 2 : Movie Poster

2 Application 3 : Note Taker (version 1)

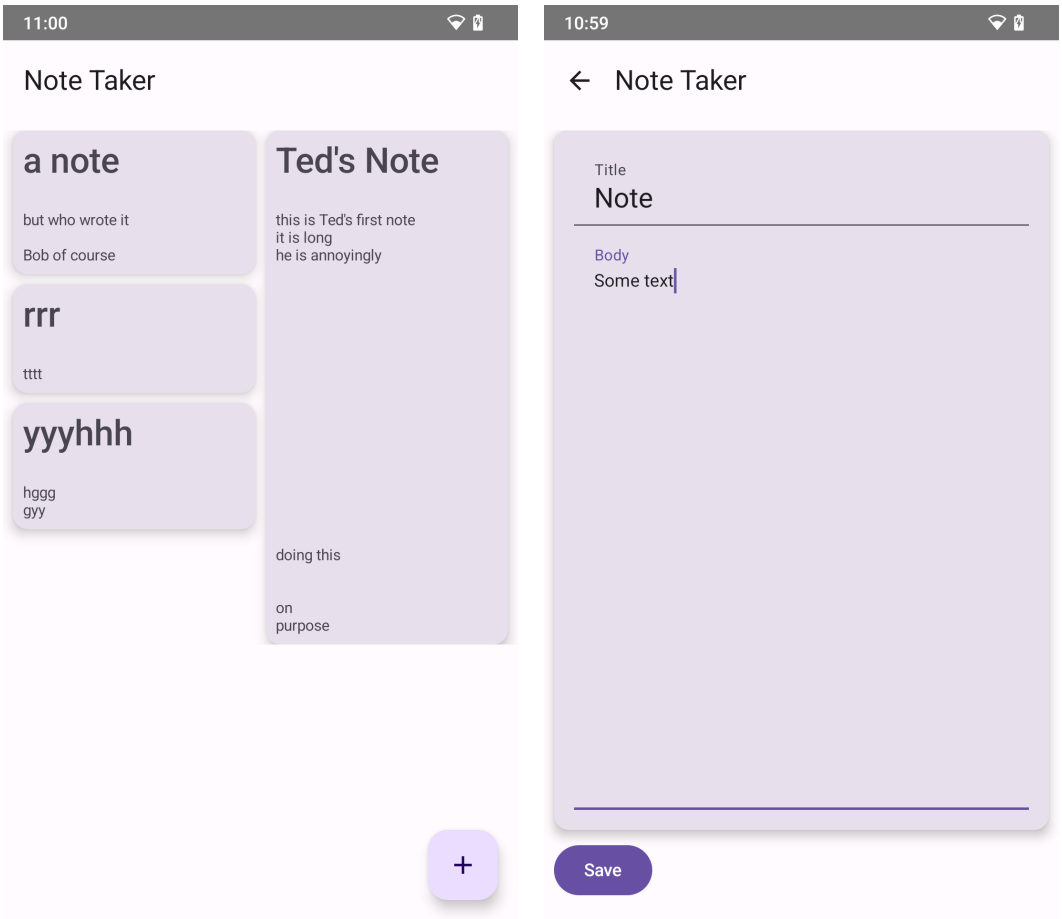
Introduction

A two screen application based upon Google keep - but much simpler..., allowing the user to:

1. View in a scrolling list the different notes made
2. Click a button to create a new note
3. Click a note to edit it
4. Either save or discard the created / modified note

However, we will not be able to persist the data (see later application updates)

Screenshots



Competences and Techniques

- Using Compose
- Model - View - View-Model (MVVM)
  - with State
- Compose Navigation API

Instructions

Whilst following these instructions, it will be sometimes necessary to fix missing imports[1], or add annotations to remove syntax warnings. This is left to the student to do.

Initial Project Creation

Within android studio, create a new project using the Empty Activity template for Phone and Tablet.

Name: Note Taker

Package name: fr.eseo.ld.<initials>.notes

Minimum SDK: API 24

Build configuration language: Kotlin DSL

Once Android Studio has finished configuring the project, launch the project to verify that the application works - you should just see the greeting screen.

Initial Data Structure

In this first iteration of the Note Taker application, we will use a Kotlin list of instances of a data class, to store the different notes. In the later iterations, we will use a local database and also a cloud based database for the data storage.

Each note, will be identified by an id, the title and body of the note, the name of the author, along with the creation and last modification dates.

1. Create the package `model`
2. Within this package create a Kotlin *data class* `Note` which will represent a note in our application
3. add read only properties to the primary constructor:
    - `id : String`
    - `author : String`
    - `creationDate : Date`[2]
    - `modificationDate : Date`
4. add read-write properties to the primary constructor:
    - `title : String`
    - `body : String`

Initial Summary Screen

The user of the note taker application, when they start it, will first have a screen that will show all the different notes that have already been created. As the number of notes which potentially can be created is rather large (up to the memory limits imposed by the system), there will possibly be more information than can be displayed in one screen. To get around this constraint, the user will be able to scroll to see the different notes. The way in which we can configure this in Compose is a lot easier than with the original Views api - We will create three different versions to show some of the different ways it is possible to do this.

Along with the scrolling list (where the different list items will be clickable, to be able to edit them), the user will be presented with a button to be able to create a new note. This button, will *float* above the list of notes.

As with the previous example, we will create a new Kotlin file to declare the composables, however as we are going to be creating multiple (here only two) screens, we will first create a sub-package to store the Kotlin code in.

1. Create the package `screens` in the `ui` package.

[1] Unfortunately, sometimes Android Studio will highlight that an import is missing, but not be able to provide the correct choice in its quick-fix pop-up menu.

[2] `java.util.Date`

2. Create a new Kotlin file `SummaryScreen` within this package.

The basic structure will start off similar to the previous example, a `Scaffold` with an `AppBar` displaying the application's title. To this we will add a `FloatingActionButton` - with no functionality for the moment.

1. Create the composable functions in `SummaryScreen` to be able to create and preview the screen which will display the title within the `AppBar`.
2. Within the `Scaffold` composable, between the topBar and the content properties, add the following code which will add a `FloatingActionButton` in the bottom right-hand corner of the screen:

```
floatingActionButton = {
        FloatingActionButton(onClick = {}) {
            Icon(imageVector =  Icons.Default.Add,
                contentDescription = "Add␣Notes")

        }
    }
```

3. Extract the `"Add␣Notes"` string to make it a string resource
4. Check the preview appears correctly within Android Studio.

The final step in the initial summary screen is to be able to display a list of notes (we will modify the preview method to create a Kotlin list of instances of `Note`, using dummy data).

We could use a Column composable as in the previous example to display the different notes, however for this application, where we are not sure how many items are to be displayed (and there could be *many* items displayed), it would not be the most efficient way of doing this. Instead, we will use a `LazyColumn`, which will optimize the memory use.

Another advantage of a LazyColumn, is that it allows us to pass a list of items to it, and declare a composable function to describe how each item will be displayed.

To be able to implement this, we need to follow three steps:

1. Create the Composable function, which will have as a parameter an instance of our data class `Note`, and will display the information
2. Create the Composable function, which will take the `List` of `Note`s, and that will display the contents of the list within a `LazyColumn` composable, using the previous Composable function to declare how each item will be displayed.
3. Use the second function as the content of our `Scaffold` composable.

Displaying a single Note To display a single note, we will use a `Card` composable. The `Card` composable is one of the fundamental bases of the material design philosophy behind modern Android layouts. By default, a card is a rectangular widget which can have an *elevation*. We need to make sure that the elevation is lower than that of the `FloatingActionButton`.

1. Create a Composable function `SummaryItem` which takes an instance of `Note` as its first parameter and a modifier as its second parameter) within the `SummaryScreen` Kotlin file.
2. Inside this function, declare a `Card` composable using the following code:

```
Card(
    elevation = CardDefaults.cardElevation(defaultElevation = 6.dp)
){

}
```

3. Add to the card composable
   - a `Text` composable to display the note's title:

```
Text(
    text = note.title,
    fontSize = MaterialTheme.typography.headlineMedium.fontSize,
    fontWeight = FontWeight.Medium,
    modifier = Modifier
        .fillMaxWidth()
        .padding(8.dp)
)
```

- a `Spacer` composable:

```
Spacer(modifier = Modifier.height(8.dp))
```

- a `Text` composable to display the note's body:

```
Text(
    text = note.body,
    fontSize = MaterialTheme.typography.bodySmall.fontSize,
    fontWeight = FontWeight.Normal,
    modifier = Modifier
        .fillMaxWidth()
        .padding(8.dp)
)
```

Display a list of items in a LazyColumn The `LazyColumn` is a composable, that will display a list of items, which is scrollable and memory efficient.

1. Create the composable function `SummaryList` which will take a List of notes and a modifier as its two parameters, which will be used to display a list of notes.
   `fun SummaryList(notes : List<Note>, modifier : Modifier = Modifier)`
2. Use a `LazyColumn` composable to display the list of items:

```
LazyColumn(
    modifier = modifier.padding(8.dp),
    verticalArrangement = Arrangement.spacedBy(8.dp)
){
    items(notes){
        note ->
            SummaryItem(note, modifier = modifier)
    }
}
```

Adding the LazyColumn to the Scaffold composable Modify the content of the `Scaffold` composable to be:

```
content = {innerPadding ->SummaryList(modifier = Modifier.padding(innerPadding))}
```

Update the preview method Replace the annotated preview method with the following code to be able to preview how the screen will be displayed. (Note here we are bypassing the `SummaryScreen` function by implementing its functionality directly in the preview method, as this is the *easiest* way of ensuring that the class compiles and the preview can be displayed). *Note, you may have to modify certain names in this code*

```
@Preview(showSystemUi = true)
@Composable
private fun PreviewNotesSummaryScreen(){
    NoteTakerTheme {
        Scaffold(
            topBar = {
                val layoutDirection = LocalLayoutDirection.current
                NotesSummaryAppBar(modifier = Modifier
                    .fillMaxWidth()
```

```
                    .padding(
                        start = WindowInsets.safeDrawing
                            .asPaddingValues()
                            .calculateStartPadding(layoutDirection),
                        end = WindowInsets.safeDrawing
                            .asPaddingValues()
                            .calculateEndPadding(layoutDirection)
                    )
                    .background(MaterialTheme.colorScheme.primary)
                )
            },
            floatingActionButton = {
                FloatingActionButton(onClick = {}) {
                    Icon(imageVector =  Icons.Default.Add,
                        contentDescription = "Add␣Notes")

                }
            }
        ){
            innerPadding ->
            val dummyNotes= listOf(
                Note(
                    id="1",
                    title="My␣first␣note",
                    body = "Not␣very␣interesting",
                    author = "Bob",
                    creationDate = Date(),
                    modificationDate = Date()
                ),
                Note(
                    id="2",
                    title="My␣second␣note",
                    body = "Not␣very␣interesting\n\but\n␣a␣lot␣longer\nthan␣other␣notes.",
                    author = "Bob",
                    creationDate = Date(),
                    modificationDate = Date()
                ),
            )
            SummaryList(notes = dummyNotes, modifier = Modifier.padding(innerPadding))
        }
    }
}
```

Once the preview is visible, create more items in the list, so that there are more items than screen space, recheck the preview. Notice how the different cards are correctly sized for their contents.

Second Style Summary Screen

The Google Keep application presents the different notes in a grid and not a single column.

The `LazyGrid` composable allows us to do this. To implement the grid rather than the column, we need to make three modifications to our `SummaryList` method:

1. change the component from `LazyList` to `LazyVerticalGrid`[3]
2. define the number of columns in the grid `columns = GridCells.Fixed(2)`
3. define the horizontalArrangement: `horizontalArrangement = Arrangement.spacedBy(8.dp)`

Final Style Summary Screen

By using the `LazyVerticalGrid`, we have an application similar to GoogleKeep, however each new row of notes are aligned at the top. If the height of each card is different, it could result in some wasted screen space. There is a component that can be used to remove this problem: A `LazyVerticalStaggeredGrid`. To be able to use one of these, make the three following changes:

1. modify the component `LazyVerticalGrid` to `LazyVerticalStaggeredGrid`
2. Replace the verticalArrangement by: `verticalItemSpacing = 8.dp`

[3] We could also make a horizontally scrolling grid

3. replace the `GridCellsFixed` to `StaggeredGridCells`

The layout of the first screen is now complete. The next step will be to create the second screen, before adding the functionality to each.

Initial Detail Screen

The second screen (the Details Screen) will allow the user to either create or modify a note. There will be two text fields to allow editing the title and the body of the note, a save button which will store any modifications, or a back button in the AppBar to return to the summary screen discarding the changes.

Up until now, we have created our own AppBar, for this screen we will use a composable already in the material library, the `TopAppBar`. This composable has space for a navigation icon (back arrow, or hamburger), title and space for one or more icons for tasks (if there are too many tasks, a menu is automatically created).

Under the `TopAppBar`, there will be a `Card` composable, that allows details to be added, or edited. In this first version, we will finish the layout with a save button.

Once more, at the moment, we are concentrating on displaying the interface, in a later section we will add the functionality.

Creating the TopAppBar

1. Create the `DetailsScreen` Kotlin file in the `screens` package.
2. Add the `DetailsScreen` composable function, add the `Surface` and `Scaffold` composables as previously.
3. For the Scaffold's topBar property, create a `TopAppBar` composable:

```
topBar = {
    TopAppBar (
        title = {
            Text(text = stringResource(id = R.string.app_name))
        },
        navigationIcon = {
            IconButton(onClick = {}){
                Icon(imageVector = Icons.AutoMirrored.Filled.ArrowBack,
                    contentDescription = "back"
                )
            }
        }
    )
}
```

4. Add the `@OptIn(ExperimentalMaterial3Api::class)` to the method, as the `TopAppBar` composable is new in the current version of the material api and its declarations and/or functionalities could possibly change.
5. Declare the content of the Scaffold's topBar as an empty `Text` composable, to remove the error.
6. Create a preview function to display the `DetailsScreen`.
7. Modify the colour of the `TopAppBar` to use the *primary* and *onPrimary* colours of the application's theme:

```
colors = TopAppBarsColors(…)
```

Create the content for the Details Screen The content for the `DetailsScreen` will be a `Column` composable, containing a `Card` composable to display the details and a save `Button` composable.

1. Create the `DetailsContent` composable function, declare two parameters, the first `existingNote : Note?` and the second `innerPadding : PaddingValues`
2. Add the following code to create the layout:

```
Column(
    modifier = Modifier
        .padding(innerPadding)
        .padding(8.dp)
        .fillMaxSize()
) {
    // Card containing the title, spacer, and body text field
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .weight(1f),  // Make the card take up the remaining space
        elevation = CardDefaults.cardElevation(defaultElevation = 5.dp)
    ) {
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(8.dp)
        ) {
            TextField(
                value = existingNote?.title ?: "",
                singleLine = true,
                textStyle = MaterialTheme.typography.titleLarge,
                onValueChange = {},
                modifier = Modifier.fillMaxWidth(),
                label = { Text(text = "Title") },
                readOnly = if (existingNote == null || existingNote.author == "Bob")
                    false
                else
                    true
            )

            Spacer(modifier = Modifier.height(8.dp))

            TextField(
                value = existingNote?.body ?: "",
                singleLine = false,
                textStyle = MaterialTheme.typography.bodyMedium,
                onValueChange = {},
                modifier = Modifier
                    .fillMaxWidth()
                    .weight(1f),
                label = { Text(text = "Body") },
                readOnly = if (existingNote == null || existingNote.author == "Bob")
                    false
                else
                    true
            )
        }
    }

    Spacer(modifier = Modifier.height(16.dp))

    if (existingNote == null || existingNote.author == "Bob"){
        Button(
            onClick = { /* Handle click */ },
            modifier = Modifier
                .align(Alignment.CenterHorizontally)
                .padding(bottom = 8.dp)
        ) {
            Text(text = "Save")
        }
    }
}
```

3. Replace the content of the `Scaffold` composable with a call to this function. passing **`null`** as the first argument and `innerPadding` as the second.
4. Verify that all the composables are visible in the preview.
5. Call the preview function, using an instance of the class `Note` (with and without the author being ”Bob”, to verify that the button appears or disappears correctly.

Final Style Details Screen

The save button could be placed on the AppBar, freeing up space - which in the v2 and v3 will allow us to display more information - author, dates, ...

1. Delete the `Spacer` and `Button` composables within the `Card` from the `DetailsContent` composable function.
2. Add a parameter: `existingNote : Note? = null` to the `DetailsScreen` function
3. add the `actions` property to the `TopAppBar` composable:

```
actions = {
    if(existingNote==null || existingNote.author == "Bob") {
        IconButton(onClick = { /* do something */ }) {
            Icon(
                imageVector = Icons.Filled.Done,
                contentDescription = "Localized description"
            )
        }
    }
}
```

4. modify the call to `DetailsContent()` so that the first argument is the `existingNote` instance that was passed as an argument to the `DetailsScreen()` method.
5. Modify the preview method to verify the correct display of the elements in the user interface.

Adding Navigation and View-Model

Now that we have the two screens, we need to be able to:

1. Navigate correctly between them
2. Modify the State of the components based upon the model

To be able to implement this functionality, we are going to use the compose navigation api and the view-model api.

Add the dependencies

1. Modify the `[versions]` section of `libs.versions.toml`, adding: `navigationCompose = "3.7.7"`
2. Modify the `[libraries]` section of `libs.versions.toml`, adding:

```
androidx-navigation-compose = {group = "androidx.navigation",
    name = "navigation-compose", version.ref="navigationCompose"}
androidx-lifecycle-viewmodel-compose = {group ="androidx.lifecycle",
name = "lifecycle-viewmodel-compose", version.ref="lifecycleRuntimeKtx"}
```

3. Modify the `dependencies` section of |build.gradle.kts(Module:app)|, adding:

```
    implementation(libs.androidx.navigation.compose)
    implementation(libs.androidx.lifecycle.viewmodel.compose)
```

4. Re-sync the project to download the correct dependencies.

Creating the View-Model The view-model in this first iteration of the Note Taker application will contain a list of `Notes` as `State`, along with the current `Note` (again as a `State`) along with the functions that allow to recuperate a `Note` from its `id` and to add or update a given `Note`.

1. In the `ui` package create a new package called `viewmodels`
2. Within this new package create the Kotlin class `NoteTakerViewModel`, sub-class of `androidx.lifecycle.ViewModel`

3. Create the properties for the `State` of the list of all the notes:

```
private val _notes = MutableStateFlow<List<Note>>(emptyList())
val notes : StateFlow<List<Note>> = _notes.asStateFlow()
```

4. Create the properties for the `State` for a single Note:

```
private val _note = MutableStateFlow<Note?>(null)
val note : StateFlow<Note?> = _note.asStateFlow()
```

5. Create the function that will add or update a `Note`:

```
fun addOrUpdate(note : Note) {
    _notes.value = if(_notes.value.any{it.id == note.id}) {
        _notes.value.map(if(it.id==note.id) note else it}
    }
    else {
        _notes.value + note
    }
}
```

6. Create the function that will recuperate a `Note` from the list from its `Id`:

```
fun getNoteById(noteId : String) {
    _note.value = _notes.value.find {it.id == noteId}
}
```

Creating a Navigation Graph The navigation api, provides the functionality to navigate between screens, passing where needed different parameters.

The compose navigation api is based upon a number of concepts:

Destination Can be thought of as a Screen (or possibly part of a screen) to be displayed in the application.

Route Uniquely identifies a destination and the data that is required for it

Graph a data structure that defines all the destinations within the application and how they are joined together

Controller manages the navigation between the different destinations in the navigation graph, including handling correctly the back stack (where to go when the back button / gesture is activated).

Host A UI element (could be the complete UI, or part of the UI) that contains the current Destination. On navigation, the application swaps the content of the host to the new destination.

The different destinations are defined by Strings. To add a simple compile time check to the navigation code, in our implementation, we are going to create an enumeration which will be used to limit the acceptable values[4].

1. In the `ui` package, create the `navigation` sub-package
2. In this package, create a Kotlin *enum class* `NoteTakerScreens` with a `String` read-only parameter called `id`
3. Add the following items:

```
SUMMARY_SCREEN("summary_screen"),
DETAILS_SCREEN("details_screen")
```

The next step is to create the class which will declare the `NavController` and `NavHost`.

1. In the `ui` package create the Kotlin file `NoteTakerApp`
2. Create a composable function `NoteTakerApp`, containing a read-only property called `viewModel` which will be initialized as an instance of our `NoteTakerViewModel` class.

[4] avoiding possible typos in the code for example

3. Create a read-only property called `navController`, using the `rememberNavController` function, so that any time this function is called for recomposing, the same navController will be used, and not a new instance :

```
val navController = rememberNavController()
```

4. Create the `NavHost`, linking to the declared `navController` and declaring that the first screen to be displayed will be the summary screen:

```
NavHost(
    navController,
    startDestination = NoteTakerScreens.SUMMARY_SCREEN.id
) {

}
```

The next step is to modify the contents of the Kotlin file `SummaryScreen`, so that we can pass the navigation controller and the view-model to the screen.

1. Add two parameters to the `SummaryScreen` function:

```
navController : NavController,
viewModel : NoteTakerViewModel = viewModel()
```

2. In the body of the function, create a read-only variable `notes` that will recuperate as a `State` the list of notes from the view-model:

```
val notes by viewModel.notes.collectAsState()
```

Modify the Kotlin *file* `DetailsScreen`, to link the navigation and view-Model.

1. Modify the parameters of the `DetailsScreen` function:

```
fun DetailsScreen(
    navController : NavController,
    noteId : String,
    viewModel : NoteTakerViewModel = viewModel()
)
```

2. create the read-only variable `existingNote`

```
val existingNote by viewModel.note.collectAsState()
```

3. Add the following lines, which will ensure that the `existingNote` instance is correctly initialized when the composable is (re)launched:

```
LaunchedEffect(noteId){
    viewModel.getNoteById(noteId)
}
```

4. Add the other variables which are going to be used to remember the state of the model:

```
var title by remember {mutableStateOf(existingNote?.title ?: "")}
var body by remember {mutableStateOf(existingNote?.body ?: "")}
val author = "Bob"
val date = Date()
```

5. Create a private function that returns a String representing a formatted date:

```
private fun formatDate(date : Date) : String {
return SimpleDateFormat("yyyy-MM-dd␣hh:mm:ss").format(date)
}
```

6. Delete the `DetailsContent` method and replace the call to it with the following[5]:

```
Column(
    modifier = Modifier
        .padding(innerPadding)
        .padding(8.dp)
        .fillMaxSize()
        .verticalScroll(rememberScrollState())
) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .weight(1f),
        elevation = CardDefaults.cardElevation(defaultElevation = 5.dp)
    ) {
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(8.dp)
        ) {
            TextField(
                value = title,
                singleLine = true,
                textStyle = MaterialTheme.typography.titleLarge,
                onValueChange = {title = it},
                modifier = Modifier.fillMaxWidth(),
                label = { Text(text = "Title") },
                readOnly = if (existingNote == null || existingNote?.author == "Bob") false else true
            )

            Spacer(modifier = Modifier.height(8.dp))


            TextField(
                value = body,
                singleLine = false,
                textStyle = MaterialTheme.typography.bodyMedium,
                onValueChange = {body = it},
                modifier = Modifier
                    .fillMaxWidth()
                    .weight(1f),
                label = { Text(text = "Body") },
                readOnly = if (existingNote == null || existingNote?.author == "Bob") false else true
            )
            Spacer(modifier = Modifier.height(8.dp))
            Row(
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.SpaceBetween
            ){
                Text(text = "Created By")
                Text(text = existingNote?.author ?: "")
            }
            Row(
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.SpaceBetween
            ){
                Text(text = "Creation Date")
                Text(text = formatDate(existingNote?.creationDate ?: date))
            }
            Row(
                modifier = Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.SpaceBetween
            ){
                Text(text = "Last modification Date")
                Text(text = formatDate(existingNote?.modificationDate ?: date))
            }

        }
    }
}
```

7. Modify the `onClick` property of the `IconButton` in the actions so that it updates the view-model:

[5] Before deleting, a lot of this could be recuperated from the DetailsContent method

```
onClick = {
    val newNote = Note(
        creationDate = existingNote?.creationDate ?: date,
        id = existingNote?.id ?: System.currentTimeMillis().toString(),
        title = title,
        body = body,
        author = author,
        modificationDate = date
    )
    viewModel.addOrUpdate(newNote)
}
```

The last step is to finish creating the navigation graph and correctly handling the navigation between screens.

1. Modify the `NoteTakerApp` function, to add the possible routes:
    - to the summary screen
    - to and from the details screen
2. Add a composable route to the summary screen inside the `NavHost` declaration:

```
composable(
    NoteTakerScreens.SUMMARY_SCREEN.id
) {
    SummaryScreen(navController, viewModel)
}
```

3. Add a composable route to and from the details screen:

```
composable(
    NoteTakerScreens.DETAILS_SCREEN.id+"/{noteId}",
    arguments = listOf(
        navArguments("noteId") {
            type = NavType.StringType
        }
    )
) {
    backStackEntry ->
        val noteId = backStackEntry.arguments?.getString("noteId") ?: "NEW"
        DetailsScreen(navController, noteId, viewModel)
}
```

Modify the `SummaryScreen` so that when a user clicks on the floating action button, they will navigate to the Details Screen to be able to create a new Note:

```
FloatingActionButton(onClick = {
    navController.navigate(NoteTakerScreens.DETAILS_SCREEN.id+"/NEW")
})
```

Modify the `SummaryScreen` so that when a user clicks on a card in the grid, they will navigate to the Details Screen to be able to modify an existing Note:

1. modify the `SummaryItem` method:

```
fun SummaryItem(note : Note, onClick : (String) -> Unit){
    Card(
        onClick = {onClick(note.id)},
        //...
```

2. modify the `SummaryList` method

```
fun SummaryList(
    modifier : Modifier = Modifier,
    notes: List<Note> = emptyList<Note>(),
    onClick : (String) -> Unit
){
    //...
    note ->
        SummaryItem(note, onClick = {onClick(it)}
```

3. Modify the call to the `SummaryList` method (in the `SummaryScreen` method):

```
SummaryList(modifier =  Modifier.padding(innerPadding),
    notes = notes,
    onClick = {navController.navigate(
        NoteTakerScreens.DETAILS_SCREEN.id+"/${it}"
    )}
)
```

Modify the `DetailsScreen` Kotlin *file*, to take into account that the user wishes to save the modifications made:

1. In the `onClick` property of the `IconButton` in the actions, after updating the view model, add the following line:

```
navController.navigateUp()
```

2. To navigate back to the Summary Screen without saving the modifications (using the back naviagation button on the app bar, modify the `onClick` property of the `IconButton`:

```
IconButton(onClick = {navController.popBackStack()})
```

For that the application can be tested on a real device, the code in the `MainActivity` Kotlin *class* needs to be modified. Simply call the `NoteTakerApp` constructor from within the `setContents()` block and delete the example composable code.

The application is complete, but contains some bugs. The list of notes is not persisted. If you close and reopen the app, (or just rotate the device) the notes have disappeared. Two solutions for this will be seen later in the Applications 5 and 6. Also, there is a small bug when the user wishes to edit the note. Occasionally, the note's contents are not displayed, and the user needs to use the back button and reopen the note. Again, with the solutions proposed in the later apps, this problem will disappear.