

Mobiele Apps 2: Library App

Alexander Schelfhout

# Inleiding

De app die ik heb ontwikkeld voor dit vak is een bibliotheek app. Deze app wordt gebruikt om een catalogus te maken van alle boeken in ons huis aangezien de collectie zo groot en verspreid is dat er al meerdere keren duplicaten zijn aangekocht. De app is ontwikkeld met Kotlin, Room database en jetpack compose in een mvvm-structuur.

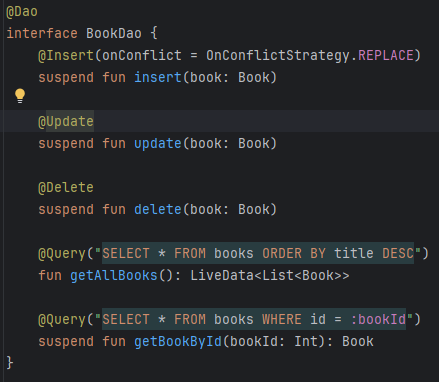
## Gebruiksaanwijzing

Wanneer je als gebruiker de app opstart zie je de titel van de app, een zoekbalk, en de headers van de tabel waarin al je boeken zullen gestoken worden wanneer je ze hebt aangemaakt. Om deze tabel te vullen druk je eerst op de floating knop met het “+”-teken. Hierna zal je doorgeleid worden naar het scherm om een nieuw boek toe te voegen aan je lijst.  
Het scherm voor een nieuw boek toe te voegen bevat drie tekstvelden om de gegevens van je boek in te vullen. Wanneer je deze velden hebt ingevuld druk je op de “save”-knop en zal je terug naar het overzicht scherm worden geleid waar je boek nu aan de lijst is toegevoegd. Als je op de pijl bovenaan drukt wanneer je in het scherm zit voor een nieuw boek toe te voegen dan ga je terug naar het overzicht. De “cancel”-knop doet hetzelfde.  
Het vuilbak icoontje dat bij het elke regel staat verwijderd onmiddellijk het boek uit de lijst.  
Wanneer je op het boek in de lijst drukt kom je in een ander scherm waar je de data van het boek kan aanpassen.  
Wanneer je een specifiek boek zoekt in de zoekbalk, zal de lijst worden gefilterd naargelang de titel of de auteur de letters bevat die je in de zoekbalk ingeeft. Om verder te sorteren kan je op de headers klikken om de lijst te sorteren op alfabetische volgorde voor titel of voor auteur.

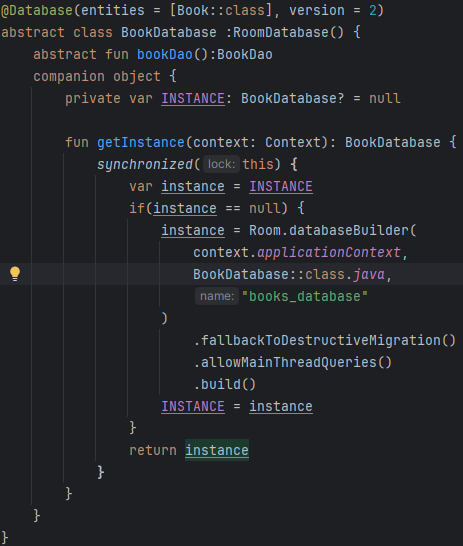
# Technische uitleg

### Database & Dao

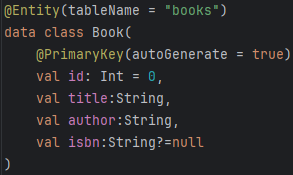
Een Dao (Data Access Object) word gebruikt in Room om de lokale database aan te spreken in plaats van rechtstreeks vanuit de code queries naar de database te verrichten. De BookDao die in mijn project word gebruikt bevat simpele methodes voor CRUD-functionaliteit: delete, insert, update, getOne, getAll. getAll geeft een LiveData object terug voor realtime updates van de boekenlijst.



De bookdatabase werd geïnstantieerd als singleton om ervoor te zorgen dat er maar één database word gecreëerd en gebruikt zoals we dat hebben gezien in MonkeyFinder.

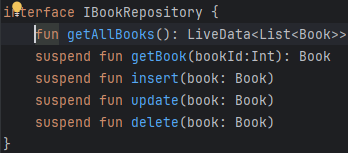


### Model: Book

Het enigste Model dat word gebruikt en ook de entity is voor de roomdatabase is de Book class. Het enigste hier dat anders is dan de MonkeyFinder die we in de les hebben gemaakt is autoGenerate, maar het enigste wat dit doet is de database automatisch een unieke id geven aan nieuwe entries.  


### Repository & IRepository

Repository klassen worden gebruikt om de dao van de database op te vragen en verrichtingen uit te voeren in de code. We gebruiken een IRepository om aan Dependency Injection en Dependency Inversion te doen.



Voor de rest implementeert de repository alle functionaliteit die de dao ook voorsziet en roept de dao van de database aan om de verrichtingen te doen.

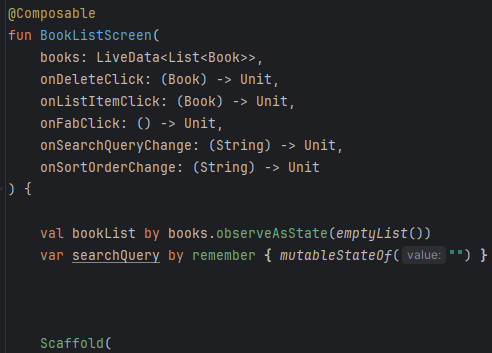


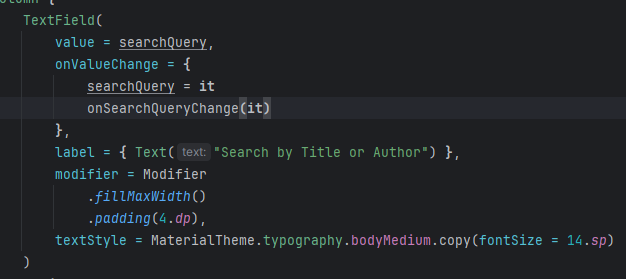
### LibraryApp

In de library app word de navigatie opgebouwd en de delen van de app die in elk scherm aanwezig zijn zoals de top en bottom appbar. Hier worden de composables ook aan de viewmodels gelinkt. De startdestination voor de navhost is het listscherm zoals dat boven in het bestand word gedeclareerd.

### BookListScreen

De BookListScreen word als eerste geladen en maakt gebruik van een LiveData object dat een Lijst van boeken bevat. Dit LiveData object wordt geobserveerd door de UI zodat het automatisch de waarde laat zien wanneer deze veranderd word bv. wanneer we een boek toevoegen. Verder bevat het nog 5 delegates voor de functionaliteiten die in de viewmodel zullen worden uitgevoerd.



De textbox bovenaan op het scherm geeft de zoekopdracht door aan het viewmodel en vuurt het onquerychanged event af. 

De tekstvlakken van de headers Titel & Author zijn tevens knoppen die het event onSortorderchanged afvuren wanneer er op gedrukt word. Dit zal dan de lijst sorteren op auteur of titel naargelang welke header op gedrukt is.

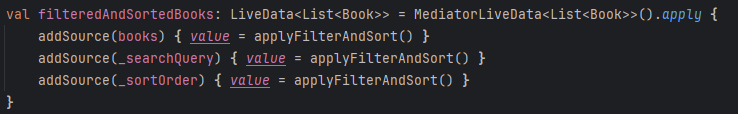


Verder maakt de Listscreen gebruik van een Lazycolumn om voor elk item in de boekenlijst die eraan word meegegeven een BookListitem aan te maken dat vervolgens gescheiden word door een lijn van het volgende item. Opmerkelijk hier is de poging om er voor te zorgen dat elk item ook een licht verschillende kleur heeft om de entries overzichtelijker en ordelijker te maken. Hiervoor maakt het gebruik van LazyListScope.indexeditems in plaats van lazylistScope.items. Op deze manier kunnen we de index van elk item gebruiken voor bewerkingen

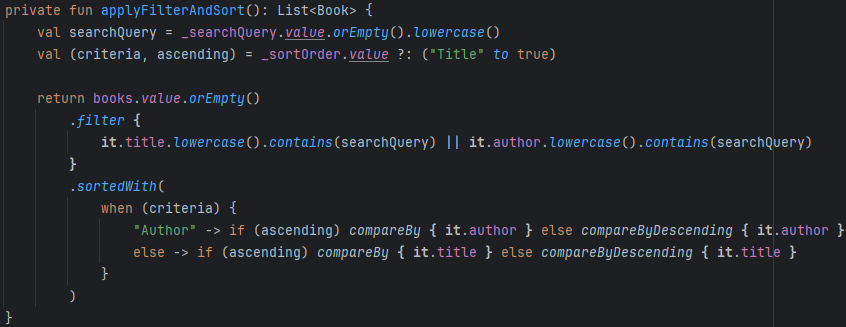
### BookListViewModel

Het Booklistviewmodel bevat alle logica die op het listscreen word uitgevoerd. Het gebruikt de repository om de data in de database aan te spreken. Het bevat twee privé eigenschappen van het type MutableLiveData. Eén om de invoer van het Zoekveld bij te houden en één om de volgorde en categorie voor het sorteren bij te houden. Verder bevat het nog twee LiveData objecten die de totale boekenlijst en de gefilterd en gesorteerde lijst van boeken bevat. Vervolgens zij er nog twee functies die de waarde van de MutableLiveData objecten aanpassen. Deze worden gekoppeld aan de delegate eigenschappen van het BookListScreen waar ze gekoppeld worden aan de sorteerknoppen in de headers en aan het zoekveld bovenaan in het BookListScreen. Het bevat ook een simpele delete functie die enkel de repository aanroept om een Book object uit de database te verwijderen.

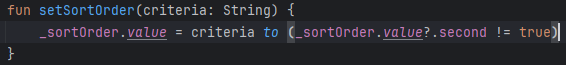
Twee onderdelen van het viewModel die verder uitgelicht moeten worden zijn het LiveData property filteredAndSortedBooks en de functie ApplyFilterAndSort.



Het Property filteredAndSortedBooks is een LiveDataObject dat geobserveerd word door de LazyColumn in het BookListScreen. Dit LiveDataObject is op zijn beurt een MediatorLiveData object dat input krijgt van de drie overige LiveDataObjecten die hierboven al waren besproken. Door addSource wordt de applyFilterAndSort methode aangeroepen wanneer een van de waardes van de drie sources veranderd.

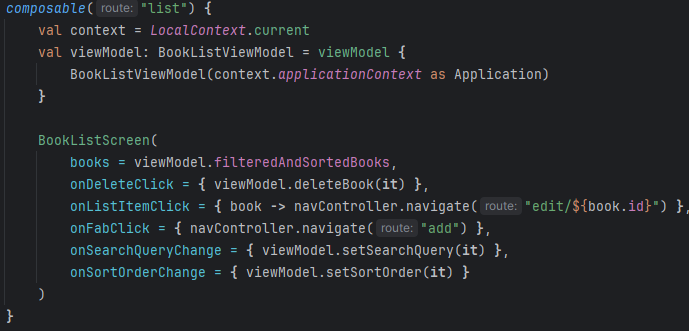


De MutableLiveData \_searchquery wordt aangepast wanneer de functie setSearchQuery wordt aangeroepen. Dit gebeurt wanneer er iets wordt ingegeven in de zoekbalk op het BookListScreen. De MutableLiveData \_sortOrder is een tuple; een enkele variabele die twee waardes van verschillende soort bevat. Dit object wordt aangepast door de functie SetSortOrder die aangeroepen wordt wanneer men op de headers drukt op het overeenkomstige scherm.



De functie die uitgevoerd word door de Headers neemt de waarde van de header (title of auteur) mee als criteria en geeft het aan de eerste waarde van de tuple. De tweede waarde is een boolean en word geïnverteerd. Het LiveDataProperty books is gewoon het resultaat van de getAllBooks functie in de repository.

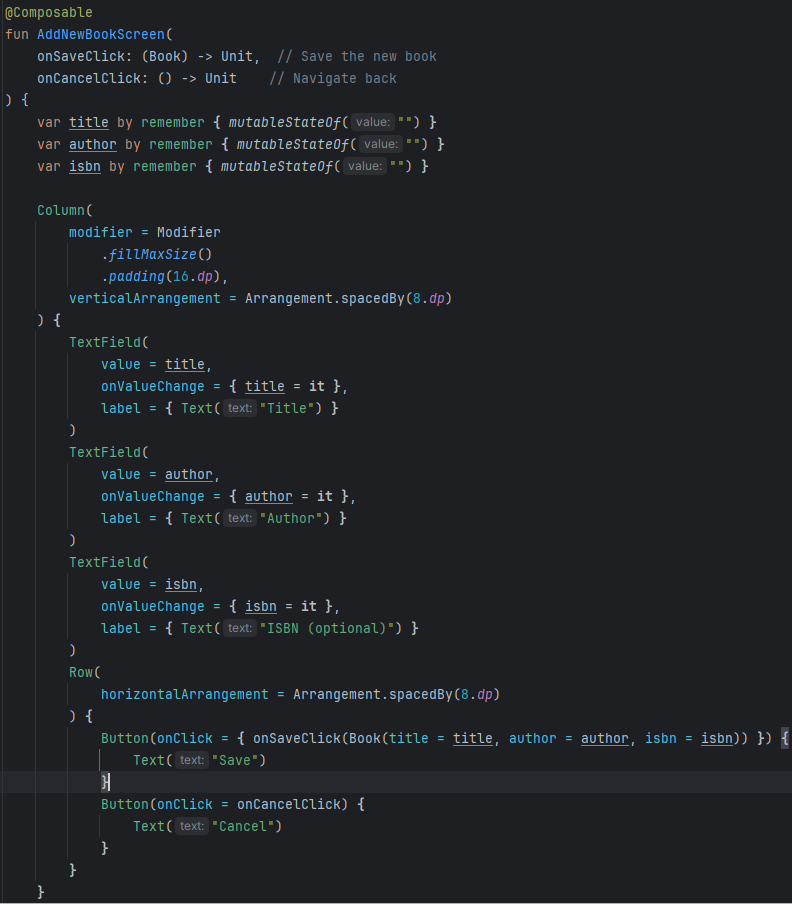
Wanneer de functie applyFilterAndSort wordt aangeroepen plaats het de LiveData objecten in een lokale variabele en geeft het een standaard waarde indien het property NULL is. De functie geeft de LiveData-lijst books terug (of een lege lijst indien books NULL is), maar past hier de functies “.filter” en “.sortedWith” op toe. De “.filter”-functie van het List<> object checkt of het object uit de lijst in zijn auteur veld OF in het title veld de searchquery bevat. Omdat \_searchquery een MutableLiveData is en de SetSearchquery functie iedere keer word uitgevoerd wanneer het onValueChanged event van het TekstField in het screen afvuurt voert dit ook applyFilterAndSort uit. Hierdoor veranderd de getoonde lijst in het screen iedere keer wanneer er een letter in het tekstveld word ingegeven. Vervolgens word de “.sortedWith” functie van het lijst-object uitgevoerd. Het vergelijkt de waarde van criteria met de String Author. Daarna checkt het de waarde van de boolean ascending.



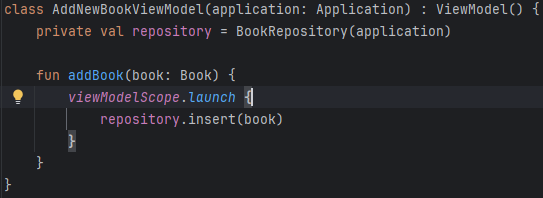
Dit is de Composable in de LibraryApp activity. Hier worden de functies en eigenschappen van het viewmodel aan de eigenschappen en delegates van het BookListScreen gekoppeld. Nog niet besproken zijn onFabClick en onListItemClick. Deze delegates zorgen er voor dat men naar het AddNewBookScreen navigeert wanneer men op de FloatingActionButton drukt, en naar het EditBookScreen wanneer men op een item in de getoonde lijst drukt. De onListItemClick delegate stuurt tevens het book object dat het event heeft afgestuurd mee naar het editBookScreen.

### AddNewBookScreen en AddNewBookViewModel

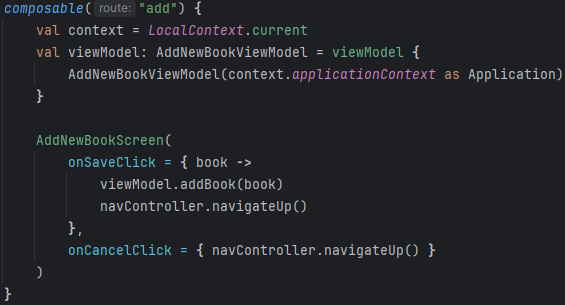
Om een nieuw boek aan te maken drukt men op de floating action button rechts onderaan.



Het AddNewBookScreen heeft drie lokale variabelen die bij aanvang lege strings zijn. Dit word geregeld door “ by remember MutableStateOf(“”)”. De drie TextFields kennen hun waarde toe aan deze variabelen wanneer de inhoud van hun field veranderd word (onValueChange). Wanneer men op de “Save”-knop drukt word er een object van het type book meegestuurd met de delegate waar de eigenschappen zijn gelinkt aan de eigenschappen die bovenaan zijn gedefinieerd.



De enigste Functie in het ViewModel is de functie die het book object dat meegestuurd word vanuit het screen toevoegt aan de database. Dit triggert opnieuw de functie applyFilterAndSort van het BookListViewModel.



De composable voegt aan de cancel- en save-knop ook nog de functie navController.navigateUp() toe zodat men terug naar het vorige scherm gaat.

### EditBookScreen en EditBookViewModel



De EditBookViewModel ontvangt een bookId die het gebruikt om de getBook functie van de repository aan te roepen. Het doet dit wanneer het geïnitialiseerd word. De getBook functie wijst het LiveData<Book> object toe aan de prive variabele \_book en vervolgens aan de book-eigenschap via de Get() methode. Verder heeft het een functie die de update methode van de repository aanroept en een die de delete methode aanroept.