

Comparator

Jetpack Composable Android App

Verheyden David

1. Inleiding.....	3
2. Doelstelling.....	3
3. Technologie	3
3.1. Composables.....	3
3.2. StateFlow.....	3
3.3. ApiCalls.....	4
4. Code.....	4
4.1. Model.....	4
4.2. Network Service	7
4.3. ViewModel.....	8
4.4. View	10
5. Besluit	15

1. Inleiding

Voor het OpleidingsProgrammaOnderdeel (OPO) "Mobiele Apps 2" kregen studenten de opdracht om een Android-app te ontwikkelen in Kotlin met behulp van Androidstudio. Voor dit specifieke project is ervoor gekozen om Jetpack Composables te gebruiken voor het ontwerpen van de gebruikersinterface. Net als bij het OPO "Mobiele Apps 1" is er gekozen voor een Model-View-Viewmodel-architectuur als softwarearchitectuur. Het doel van de app is om via een API verschillende webwinkels te doorzoeken naar hetzelfde product, zodat prijzen kunnen worden vergeleken.

2. Doelstelling

De einddoelstelling voor dit OPO is een Android Composables Applicatie Die aan de hand van een door de gebruiker gekozen zoekterm, met behulp van een API, producten op te halen bij webwinkels en deze producten weer te geven in de app. De hoofdpagina zou een scrol view moeten zijn en als op een product getikt wordt dan hoort een Details pagina getoond. Het moet ook aan de gebruiker duidelijk gemaakt worden wanneer de API-call bezig is of wanneer deze API-call gefaald is. De gebruiker moet kunnen zien van welke webshop het product afkomstig is.

3. Technologie

3.1. Composables

Jetpack Compose, een modern framework voor Android-applicaties, introduceert een declaratieve aanpak voor het bouwen van gebruikersinterfaces, waardoor het traditionele imperatieve programmeermodel wordt omzeild. In tegenstelling tot het schrijven van XML-bestanden, maakt Compose gebruik van een expressieve syntax om de UI te beschrijven, wat resulteert in code die niet alleen gemakkelijker leesbaar is, maar ook eenvoudiger te onderhouden. Door het bevorderen van herbruikbare composables, kleine bouwstenen voor de UI, biedt Compose een flexibele benadering voor het samenstellen van complexe interfaces door eenvoudige componenten te combineren. De reactieve programmeerstijl van Compose automatiseert de UI-update op basis van wijzigingen in de app-staat, waardoor het beheer wordt vereenvoudigd en de kans op fouten wordt verminderd. Composables zijn interoperabel met bestaande Android Views, waardoor een geleidelijke migratie mogelijk is. Ondersteund door krachtige tools in Android Studio en volledig geschreven in Kotlin, maakt Jetpack Compose gebruik van de voordelen van de taal, zoals uitbreidingsfuncties en type-inferentie. Deze evolutie in Android-ontwikkeling wordt steeds meer omarmd vanwege de aanzienlijke voordelen op het gebied van leesbaarheid, herbruikbaarheid en onderhoudsgemak die het biedt.

3.2. StateFlow

De concepten State, StateFlow en MutableStateFlow zijn van groot belang bij het beheren van de toestand van een applicatie in Jetpack Compose.

- **State:**

Dit verwijst naar de actuele status of toestand van een object of systeem op een specifiek moment. In de context van softwareontwikkeling vertegenwoordigt het de huidige set gegevens die de status van een applicatie beschrijven, zoals de actieve pagina of geselecteerde items.

- **StateFlow:**

StateFlow is een onderdeel van de Kotlin Coroutines-bibliotheek en biedt het een onveranderlijke stroom van waarden die in de loop van de tijd kunnen veranderen. Het is met name nuttig voor het modelleren van toestandsveranderingen waarbij notificaties van belang zijn.

- **MutableStateFlow:**

MutableStateFlow lijkt op StateFlow, maar heeft als cruciale eigenschap dat de waarde op elk moment kan worden aangepast. Dit maakt het handig wanneer actieve bijwerkingen van de toestand noodzakelijk zijn en het belangrijk is om anderen op de hoogte te stellen van dergelijke wijzigingen.

3.3. ApiCalls

Er wordt gekozen voor Retrofit voor het uitvoeren van API-aanroepen en het verwerken van de resulterende gegevens. Een representatief voorbeeld van een Retrofit API-aanroep wordt gedemonstreerd in de onderstaande interface-methode:

```
@GET("api/data/{id}")
suspend fun fetchData(@Path("id") id: String): Call<DataItem>
```

In deze methode geeft de @GET-annotatie aan dat de methode overeenkomt met een HTTP GET-aanroep naar het gespecificeerde eindpunt, met een dynamisch pad "{id}". Door gebruik te maken van @Path kan Retrofit dit dynamische pad tijdens runtime koppelen aan een specifieke waarde. Bij het aanroepen van deze methode initieert Retrofit de API-aanroep en beheert de communicatie met de server. De verkregen gegevens, zoals gedefinieerd door het returntype (Call<DataItem>), worden vervolgens overgedragen aan de applicatie voor verdere verwerking. Het gebruik van het suspend-woord in Kotlin maakt asynchrone uitvoering mogelijk, wat bijdraagt aan de ontwikkeling van reactieve en efficiënte Android-applicaties.

4. Code

4.1. Model

Er zijn twee datamodelklassen en drie UI-State modelklassen, de dataklassen dienen om op een gestructureerde manier data voor de API te verwerken. De UI-State modelklassen dienen dan weer om de staat en variabelen die in de UI weergegeven moeten worden bij te houden en door te geven aan het viewModel.

- **ApiCallData**

```
1 package com.waldorf.comparator.model
2
3 data class APICallData (
4     var searchString: String?,
5     val regions: Array<String> = arrayOf("BE", "NL", "DE", "FR", "JP", "COM")
6 )
```

Hierbij is searchString de zoekterm om de API-call te maken en regions is de array met de mogelijkewaarden voor de API om webwinkels te definiëren, om de gebruiker zo eventueel de mogelijkheid te geven om bepaalde webwinkels te selecteren of uit te sluiten.

- SearchItem

```
± BeWaldorf
@Serializable
data class SearchItem(
    val name: String,
    var price: String? = "€?,??",
    @SerializedName(value = "image_link")
    val imageLink: String,
    var link: String? = null,
    @SerializedName(value = "delivery_price")
    val deliveryPrice: String? = null,
    @Transient
    var cc: String? = null
)
```

De annotatie `@Serializable` speelt een cruciale rol bij het gebruik van Retrofit om een gegevensklasse als serializable aan te duiden. Dit betekent dat objecten van deze klasse kunnen worden omgezet naar een byte-stroom voor transport of opslag. Om een aangepaste naam toe te wijzen aan een eigenschap, wordt de `@SerializedName`-annotatie gebruikt. Hiermee kan bijvoorbeeld de standaardnaam "image_link" worden vervangen. Deze functionaliteit is handig bij het matchen van de klasse met externe gegevensbronnen. Aan de andere kant, wanneer een eigenschap als `@Transient` is gemarkeerd, zoals in het geval van 'cc', de landscode die hier dienst doet als ID voor de webwinkel wordt deze eigenschap uitgesloten van de serialisatie. Dit betekent dat de eigenschap niet wordt opgenomen bij het omzetten naar een byte-stroom en dus ook niet wordt teruggegeven in de JSON van de API-calls.

- ApiUIState

```
sealed interface ApiUIState{
    ± BeWaldorf
    data class Success(val itemList: List<SearchItem>): ApiUIState
    ± BeWaldorf
    object Error      : ApiUIState
    ± BeWaldorf
    object Loading   : ApiUIState

    ± BeWaldorf
    object Empty : ApiUIState
}
```

De implementatie van `ApiUIState` heeft tot doel om verschillende API-interactiestaten gestructureerd en veilig te modelleren. Dit wordt bereikt door het gebruik van een sealed interface met drie objecten (`Error`, `Loading` en `Empty`) en een dataklasse (`Success`). Door het gebruik van een sealed interface worden alleen de gespecificeerde klassen toegestaan, waardoor alle mogelijke staten exhaustief worden behandeld. De `Success` dataklasse bevat een lijst van `SearchItem`-objecten en vertegenwoordigt een succesvolle API-oproep, waarbij relevante gegevens worden vastgelegd. De objecten `Error`, `Loading` en `Empty` fungeren als singleton-instanties en geven respectievelijk fouten, laadstatussen en lege resultaten aan. Het gebruik van sealed classes en objecten bevordert consistentie en eenvoud in de implementatie, wat de leesbaarheid vergroot en de kans op fouten verkleint door exhaustiviteit te waarborgen bij het afhandelen van alle mogelijke gevallen.

- HomeScreenUiState

```
± BeWaldorf
data class HomeScreenUiState(private val flag:Boolean
) {
    private var _homeFlag = MutableLiveData<Boolean>( value: true)
    val homeFlag: LiveData<Boolean> = _homeFlag
    ± BeWaldorf
    init {
        | _homeFlag.value = flag
    }
}
```

In deze code is een dataklasse genaamd HomeScreenUiState geïmplementeerd met als doel het beheren van de status van de gebruikersinterface op het startscherm. Deze klasse bevat een private eigenschap genaamd "flag", die de initiële waarde van de UI-status aangeeft, namelijk of de container de API-call interface of de details interface zal weergeven. Daarnaast wordt een MutableLiveData object geïnitieerd met een boolean-waarde genaamd "_homeFlag". Dit object wordt vervolgens blootgesteld als een LiveData-eigenschap genaamd "homeFlag". De initiële waarde van "_homeFlag" wordt ingesteld op basis van de meegegeven "flag" in de constructor. Deze implementatie maakt gebruik van het Observer-pattern in Android, waardoor andere delen van de applicatie de "homeFlag" LiveData kunnen observeren om wijzigingen in de UI-status te detecteren. Dankzij deze structuur wordt een efficiënte en gestandaardiseerde manier geboden om de UI-status te volgen en bij te werken.

4.2. Network Service

```

private const val BASE_URL : String = "https://waldorfscomparator.azurewebsites.net/api_search/"
private val json = Json { coerceInputValues = true }
private val retrofit = Retrofit.Builder()
    .addConverterFactory(json.asConverterFactory("application/json".toMediaType()))
    .baseUrl(BASE_URL)
    .build()

@BeWaldorf
interface ComparatorApiService {

    @BeWaldorf
    @GET("/{countryCode}/{searchTerm}")
    suspend fun getItem(
        @Path("countryCode") countryCode: String,
        @Path("searchTerm") searchTerm: String?
    ): List<SearchItem>
}

@BeWaldorf
object ComparatorApi {
    val retrofitService : ComparatorApiService by lazy {
        retrofit.create(ComparatorApiService::class.java)
    }
}

```

Deze code bevat een implementatie van een Retrofit API-service om zoekitems van een externe server op te halen. De `BASE_URL` geeft de basis-URL van de API aan, en de `Json`-instantie is geconfigureerd om JSON-gegevens te verwerken. De Retrofit-instantie wordt gemaakt met een aangepaste JSON-converter en de opgegeven basis-URL. De `ComparatorApiService`-interface definieert de API-aanroepen, waarbij de `getItem`-methode een asynchrone aanroep is om zoekitems te verkrijgen op basis van een landcode en zoekterm.

Het `ComparatorApi` singleton-object biedt toegang tot de Retrofit-service via de `retrofitService`-eigenschap, die geïmplementeerd is als een lazy-initialized eigenschap. Deze implementatie zorgt voor een efficiënte en uitgestelde instantiatie van de Retrofit-service, wat resources bespaart totdat deze daadwerkelijk gebruikt wordt. De code is gestructureerd op een overzichtelijke en herbruikbare manier om de configuratie van Retrofit, JSON-verwerking en API-servicefunctionaliteit te organiseren, waardoor het onderhoud en de uitbreidbaarheid van de code bevorderd worden.

4.3. ViewModel

```

± BeWaldorf
class HomeScreenViewModel: ViewModel() {
    private val _uiState = MutableStateFlow<HomeScreenUiState>(flag: true)
    val uiState: MutableStateFlow<HomeScreenUiState> = _uiState
}

± BeWaldorf
class SearchBarViewModel(val api_vm: ApiViewModel) : ViewModel() {
    private val _uiState = MutableStateFlow<SearchBarUiState>()
    val uiState: StateFlow<SearchBarUiState> = _uiState.asStateFlow()

    ± BeWaldorf
    fun updateSearchTerm(searchText: String) {
        val isSearching = searchText.isNotEmpty()
        _uiState.value = SearchBarUiState(currentSearch = searchText, isSearching = isSearching)
        Log.d(tag: "SearchBar", msg: "Current search term: $searchText")
        api_vm.setCallData(searchText)
    }
}

± BeWaldorf
class SearchItemViewModel(private var testFlag: Boolean? = false) : ViewModel() {
    private var _searchItem: SearchItem? = null
    ± BeWaldorf
    val searchItem: SearchItem? get() = _searchItem

    ± BeWaldorf
    init {
        if (testFlag == null || testFlag == true) {
            this._searchItem = SearchItem(
                name: "1", price: "€20.00",
                imageLink: "https://fastly.picsum.photos/id/910/536/354.jpg?hmac=tL0q",
                link: "https://fastly.picsum.photos/id/910/536/354.jpg?hmac=tL0qOtDz3"
            )
        }
    }

    ± BeWaldorf
    fun setSearchItem(searchItem: SearchItem) { ... }

    ± BeWaldorf
    fun getImageUrl(): String? { ... }

    ± BeWaldorf
    fun getTitleString(): String? { ... }

    ± BeWaldorf
    fun getPriceString(): String? { ... }

    ± BeWaldorf
    fun getCcString(): String? { ... }

    ± BeWaldorf
    fun getLinkString(): String? { ... }

    ± BeWaldorf *
    private fun removeDotBetweenNumbers(inputString: String?): String {
        if (inputString == null) return "€ not parsed"
        val string: String = inputString.replace(
            oldValue: ".", newValue: ","
        )
        Log.d(tag: "strings", string)
        return string
    }
}

```


In de ViewModels van Comparator wordt een gestructureerde aanpak gevolgd om de logica en gegevensbeheer te organiseren. Neem bijvoorbeeld de `SearchItemViewModel`, die is ontworpen om de weergave van individuele items vanuit de API te beheren. De initiële staat van de ViewModel wordt ingesteld via een optionele `testFlag` in de constructor. Deze ViewModel biedt methoden om specifieke gegevens van een `SearchItem`-object op te halen en te presenteren.

De `ApiViewModel` beheert gegevens voor API-aanroepen en geeft deze door aan de UI via `MutableStateFlow`. Hier wordt de status van de API-aanroep weergegeven door middel van verschillende objecten van `ApiUIState`, zoals `Loading`, `Success` en `Error`. De `makeApiCall`-methode initieert de daadwerkelijke API-oproep en beheert de overgang tussen de verschillende UI-staten.

De `HomeScreenViewModel` gebruikt `MutableStateFlow` om de UI-status van het startscherm te volgen, en de `SearchBarViewModel` maakt gebruik van een `SearchBarUiState` om de zoekbalkstatus en bijhorende zoekterm bij te houden. De methode `updateSearchTerm` in `SearchBarViewModel` actualiseert de UI en initieert tevens een bijwerking van de gegevens via de bijbehorende `ApiViewModel`, wat leidt tot een georganiseerde stroom van gegevens tussen de verschillende ViewModels.

Dit ontwerp maakt gebruik van `Android Architecture Components` en `Kotlin Coroutines` om asynchrone taken te beheren, en de gedefinieerde ViewModels bieden een gestandaardiseerde structuur voor het beheer van gegevens en de communicatie tussen verschillende onderdelen van de applicatie.

4.4. View

```

± BeWaldorf
class MainActivity : ComponentActivity() {
    private val homeScreenVM: HomeScreenViewModel = HomeScreenViewModel()
    private val apiVM: ApiViewModel = ApiViewModel()
    private val _homeScreenView: HomeScreenView = HomeScreenView(homeScreenVM, apiVM)
    ± BeWaldorf
    val homeScreenView: HomeScreenView get() = _homeScreenView
    ± BeWaldorf
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //logi
        setContent {
            homeScreenView.ActivityContent()
        }
    }
}

```

Alle weergave gebeurt vanuit de Activity: MainActivity. Hier worden een HomeScreenViewModel en een ApiViewModel aangemaakt om deze ViewModels hierna mee te geven aan HomeScreenView. Uiteindelijk wordt de content van deze activity gezet op de composable functie: “Activity Content” van homeScreenView. Deze Composable functie zal een container Composable functie laden die aan de hand van HomeStateFlag ofwel Details ofwel de zoek/API/Home Composables hoort te tonen.

```

@Composable
fun ActivityContent(){
    Container()
}

± BeWaldorf *
@SuppressLint("StateFlowValueCalledInComposition")
@Composable
fun Container() {
    when {
        homeFlagState.value!! → {
            Log.d( tag: "views", msg: "home state: ${homeFlagState.value}")
            HomeScreen()
        }
        else → {
            Log.d( tag: "views", msg: "home state: ${homeFlagState.value}")
            val itemVm: SearchItemViewModel = SearchItemViewModel( testFlag: true)
            Details(item = itemVm.searchItem!!)
        }
    }
}

```

De status van de variabele data wordt bijgehouden door de status te collecten dit wordt aangehaald in de constructeur van HomeScreenView

```

init {
    searchBarVM = SearchBarViewModel(apiVM)
    CoroutineScope(Dispatchers.Main).launch { this: CoroutineScope
        viewModel.uiState.collect { homeScreenUiState →
            homeFlagState.value = homeScreenUiState.homeFlag.value
        }
    }
    CoroutineScope(Dispatchers.Main).launch { this: CoroutineScope
        homeFlagState.collect { boolean →
            homeFlagState.value = boolean
        }
    }
    CoroutineScope(Dispatchers.Main).launch { this: CoroutineScope
        apiVM.apiUIState.collect { newState →
            apiState.value = newState
        }
    }
}
}

```

Op het einde van de Homescreen() composable wordt ApiBlock() aangeroepen, deze functie roept aan de hand van welke APIUIState geïmplementeerd is op dat moment een andere Composable functie aan:

```

@Composable
fun ApiBlock() {
    val apiState by apiState.collectAsState()

    when (apiState) {
        is ApiUIState.Success → {
            ItemListMaker((apiState as ApiUIState.Success).itemList)
        }
        ApiUIState.Loading → {
            LoadingScreen()
        }
        ApiUIState.Error → {
            ErrorScreen()
        }
        ApiUIState.Empty → {
            EmptyBlock()
        }
    }
}

```

ItemListMaker wordt als voorbeeld hier verder besproken:

```

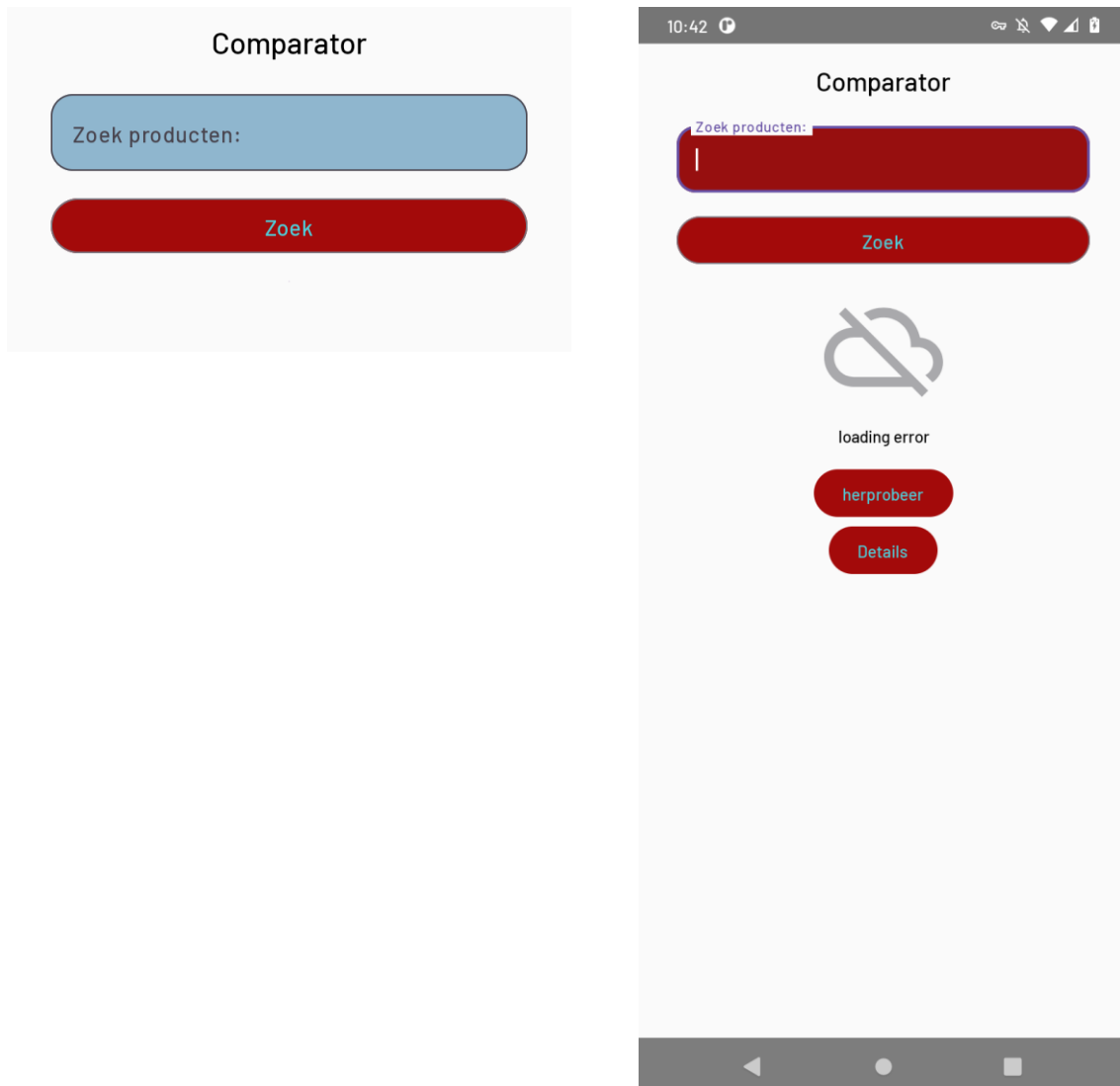
@Composable
fun ItemListMaker(itemList: List<SearchItem>, modifier: Modifier = Modifier){
    Log.d(tag: "ViewCard", msg: "API Card start")
    for( item in itemList){
        val itemVM: SearchItemViewModel = SearchItemViewModel()
        itemVM.setSearchItem(item)
        val itemView: SearchItemView = SearchItemView(itemVM)
        itemView.SearchItemCard()
    }
}

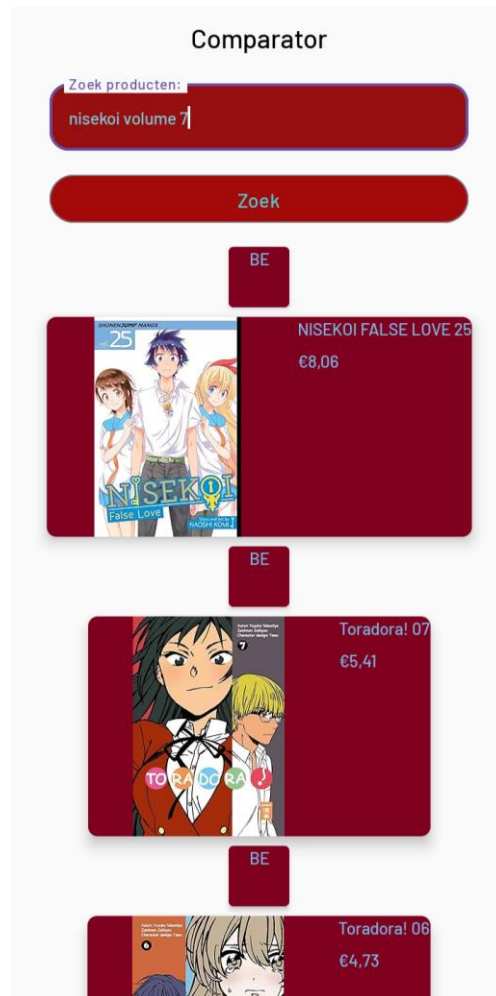
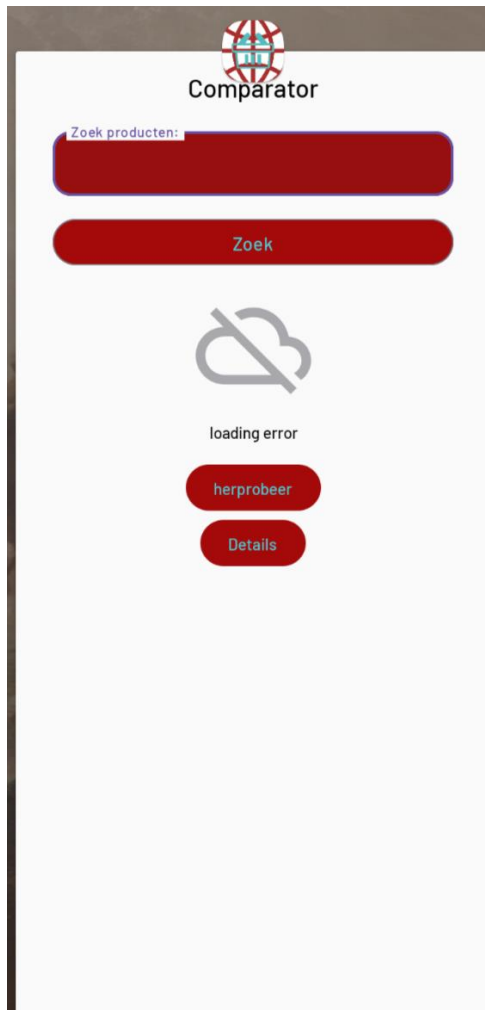
```

Comparator

Deze composable-functie, genaamd `ItemListMaker`, heeft als taak het verwerken van een lijst met zoekresultaten van een API-aanroep. Voor ieder item in de lijst wordt een nieuwe `SearchItemViewModel` gemaakt, waarin het huidige `SearchItem` wordt uitpakkt en toegewezen. Vervolgens wordt er een bijbehorende `SearchItemView` gecreëerd, waaruit de `SearchItemCard` wordt opgehaald en getoond in de gebruikersinterface. Deze functie maakt het dus mogelijk om dynamisch weergaven te genereren voor ieder item in de ontvangen lijst, waarbij de gegevens en het gedrag van ieder item worden beheerd door een apart `SearchItemViewModel` en getoond worden door de bijbehorende `SearchItemView`.

Dit resulteert in volgende schermen voor de app:






Comparator



Noodle mix of Nong Shim, Nissin, Samyang, Mama, Acecook, Kung-Fu, Ottogi with Extra Mix Brands.

\$39.98


COM



Nongshim

€ not parsed


COM



Amazon Kitchen

€ not parsed

COM



Mrs. Grass

\$23.51


10:42

Comparator

Zoek producten:

nisekoi vol7

Zoek



1 2 3 4 5 6 7 8 9 0

@ # € _ & - + () /

=\< * " ' : ; ! ?

ABC , 12 34 .

5. Besluit

De ontwikkeling van de Android-applicatie voor het OPO "Mobiele Apps 2" vereiste een grondige inzet van moderne technologieën en ontwerpprincipes. Het gebruik van Jetpack Compose als framework voor de gebruikersinterface heeft niet alleen bijgedragen aan een meer eigentijdse benadering van UI-ontwikkeling, maar heeft ook geresulteerd in code die gemakkelijker leesbaar en onderhoudsvriendelijker is. Door de implementatie van de Model-View-ViewModel-architectuur kon de logica en het gegevensbeheer gestructureerd worden georganiseerd, met behulp van ViewModel-klassen voor verschillende aspecten van de applicatie.

Jetpack Compose heeft zich bewezen als een krachtig framework vanwege zijn declaratieve benadering van UI-ontwikkeling, de bevordering van herbruikbare composables en de integratie met Kotlin, wat heeft bijgedragen aan de efficiëntie van de ontwikkeling. StateFlow en MutableStateFlow hebben een cruciale rol gespeeld bij het beheren van de applicatiestatus, waardoor een reactieve benadering mogelijk is en gegevens eenvoudig kunnen worden uitgewisseld tussen verschillende onderdelen van de app.

De keuze om Retrofit te gebruiken als HTTP-client heeft geleid tot efficiënte API-aanroepen, waarbij de verkregen gegevens gestructureerd zijn verwerkt met behulp van Serializable-klassen. Door ApiUIState te implementeren als een sealed interface met objecten en een dataklasse, is er bijgedragen aan een veilige en gestructureerde modellering van de verschillende staten van API-interactie.

Er zijn verschillende aspecten waarop Comparator verbeterd kan worden. Een van de verbeterpunten is het vervangen van HomeStateFlag door een sealed interface met een boolean, vergelijkbaar met de werkwijze van ApiUIState. Op deze manier kan de functionaliteit beter gestructureerd worden. Daarnaast kan de visuele uitstraling van de app ook verbeterd worden om een betere gebruikerservaring te bieden. Tot slot zou het interessant zijn om de implementatie van Firebase toe te voegen als een boeiende uitbreidingsoptie, waardoor de app nog meer functionaliteit kan bieden.

Als gevolg van de samenhang van deze technologieën is er een goed gestructureerde, leesbare en efficiënte Android-app ontwikkeld die voldoet aan de gestelde doelstellingen. Gebruikers hebben de mogelijkheid om producten te vergelijken via verschillende webwinkels op basis van een zoekterm, met duidelijke indicaties voor API-interactiestatussen. Het succes van dit project benadrukt de waarde van een geïntegreerde aanpak van moderne technologieën en softwarearchitecturen bij het ontwikkelen van mobiele applicaties in een educatieve context.