# OnlyPaws

Mathijs Verbeure

18 januari 2025

# Inhoudsopgave

# 1 Inleiding

De eerste vraag wanneer een project zo absurd als "een dating platform voor katten" gepresenteerd wordt is: `Waarom?` Een concreet antwoord hierop... heb ik niet. Toen het project voorgelegen werdt nam ik de beslissing om geen standard onderwerp te nemen. Wat brutaal gezegd: niks saai. Zo behoud ik mijn motivatie en valt de kraan niet zo snel dicht. Een ander punt waar ik rekening mee nam was dat dit project, indien mooi afgerond, leuk op een CV / Portfolio kan komen. Kotlin is een snel evoluerende taal die veel gebruikt wordt binnen de Google-omgeving, en zeker toegepast kan worden binnen de context van een bedrijf. Een topic kiezen die interessant is en waarover veel te bespreken valt is dus erg belangrijk; veel valt er niet te vertellen over een `To-Do` app, maar een online dating app voor poezen? Dat lokt wel aandacht!

Dit waren dus de officiële redenen voor dit project, maar er blijven er nog twee over die het meeste invloed gehad hebben op mijn beslissing.

1. Ik heb poezen graag; het zijn grappige beesten.
2. Het is een grappig concept.

*OnlyPaws* is gebaseerd op *Tinder*™, het online dating platform. Hier kunnen gebruikers profielen bekijken van andere, naar rechts swipen indien de persoon hun interesseert en naar links indien niet. Indien beide gebruikers naar rechts swipen — zal ik verder "Liken" noemen — dan is het een "Match", en kunnen ze berichten naar elkaar sturen om verder contact op te nemen.

Een vrij simpel concept dat enorm populair is geworden. Zelf had ik *Tinder* nog nooit gebruikt, enkel via een vriend of het internet de groffe lijnen begrepen. Om een app te ontwerpen hierrond moest ik uiteraard zelf eens uit proberen en notities nemen over de UI/UX. De 'testprocedure' was inzichtsvol en bracht verschillende ideeën op die ik zou proberen te integreren in mijn versie. Sommige details in de UX vielen erg hard op, zoals bij voorbeeld hoe vaak er een voorstel komt om een subscriptie te nemen die je voordelen geeft. Eens je maandelijks betaal kan je berichten sturen zonder match — via wat ze "First Impressions" noemen — bekijken wie jouw profiel geliked heeft en meer. Al zou het hilarisch zijn om tijdens de demonstratie niks te kunnen tonen omdat er betaald moet worden, dit zal ik voor overduidelijke redenen niet doen.

De economische strategieën van techgiants zijn niet te bespreken in dit verslag. Dit is een project in academische context, geld-gerichte implementaties zijn dus niet gepast. Verder nog, zelf vind ik dat bedrijven veel te gericht zijn op winst, niet op de producten die gemaakt worden; *Tinder* valt amper te gebruiken door alle reclame die ingebouwd is. Dit weiger ik op eender welke wijze te implementeren.

Deze negatieve kanten van het oorspronkelijk product zijn dus niet te vinden in dit project — en meer zelfs! Deze verschillen bespreek ik wel eens ze aan bod komen.

# 2 Verloop

Bij het openen van de app gaat Google's *Credential Manager* open. Deze onthoudt welke gegevens al gebruikt zijn om in te loggen of te registreren. Dankzij deze moet je je paswoord ook niet onthouden; het wordt allemaal veilig bewaard door Google. De pop-up laat je selecteren welk account je wilt gebruiken als, stel maar, je meerdere katten hebt. Indien er geen credentials te vinden zijn wordt je automatisch naar de Register pagina verstuurd. Verder kan je ook het Credentials menu sluiten om een gebruiker aan te maken.

Het proces om een gebruiker aan te maken neemt twee stappen, gesplitst over twee opeenvolgende paginas; eerst de belangrijke gegevens voor login, dan de informatie die andere gebruikers zullen zien. Je vult een email in, gevolgd door een paswoord. Je mag door naar de volgende stap eens deze checks goedgekeurd zijn: het emailadress mag nog niet in de database bestaan en het paswoord moet meer dan 6 karakters lang zijn. In een perfecte wereld zou een bevestigingsmail verstuurd worden, maar dat valt buiten de scope van dit project.

In de tweede pagina moet je een naam, beschrijving en afbeelding invullen, die andere gebruikers te zien krijgen. Er zijn geen beperkingen op deze, alleen ze mogen niet leeg zijn. Hier komt de scope van dit project weer in spelen: ik maak gebruik van Firebase RealTime database om alle gegevens en authenticatie op te slaan. Hier kunnen geen afbeeldingen opgeslaan worden, alleen text/JSON. Daarom zijn er op de RegisterDetails pagina twee knoppen; één om een afbeelding lokaal te openenen en één waarbij je de link naar een random foto online krijgt. Beide zijn volledig functioneel, maar de lokale afbeelding gaat door niemand buiten de huidige telefoon bekeken kunnen worden, dus raad ik niet aan.

Proficiat! U bent nu de trotse eigenaar van een OnlyPaws account en wordt doorgestuurd naar de Home pagina. Hier worden één per één andere profielen getoond. Je kan ofwel Liken ofwel Disliken, door rechts of links te klikken op de pagina. Centraal staat de foto van de kat met de naam en beschrijving bovenaan. Elke profiel die geliked wordt komt in de Favorites pagina te staan. Hier kan je alle profielen van Liked naar Dislike sturen en omgekeerd, plus de foto's bekijken door op de thumbnail te klikken. Bij het verwissellen van een profiel komt een pop-up tevoor om na te vragen of de gebruiker dit zeker wilt doen. Indien ja, wordt de operatie uitgevoerd. Indien nee sluit de pop-up en verandert er niks.

Ten laatste is er een Account pagina, waar de gebruikersgegevens aangepast kunnen worden. Een lokale foto opslaan kan nog niet, dus de profiel foto aanpassen is nog niet mogelijk, maar naam en beschrijving wel. Voer de nieuwe text in, duuw op het knopje en de gegevens worden opgeslaan. In de boven-rechtse hoek is een Log-Out knop te vinden waarmee je, uiteraard, kunt uitloggen van het account en terug naar de Login gaan.

# 3 Documentatie

## 3.1 MVVM

MVVM is een structuur waarbij er zo veel mogelijk abstractie is tussen data en display. Doorheen het process is de structuur meerdere keren vanaf nul weer begonnen, vaak omdat ik een nieuwe manier van werken gevonden had en die voorkeur gaf over de oude methode. Een van de belangrijkste concepten is het *Separation of Concerns* principe. Dit legt voor dat elk object, elke classe, elke pagina, elk data model geen kennis mag hebben van waar het gebruikt wordt, of in welke toepassing het beland. Eender welk bestand zou theoretisch geplakt kunnen worden in eender welke omgeving en werken. Een pagina zou geen fout moeten geven als een viewmodel verwisselt wordt. De structuur die ik aangenomen heb voor dit project draait om een paar begrippen die bij elke toepassing opnieuw gemaakt worden, en op een robuste manier code veiligheid behalen.

### 3.1.1 ViewModel

Viewmodels zijn de classes waarin data wordt bijgehouden. Ze doen het rekenwerk, halen data op en sturen de nodige informatie door naar de gebruikersinterface. Intern bestaan er meerdere variabelen die doorheen de levenstijd van het viewmodel geldig blijven, samen met de nodige functies om alles operationeel te krijgen. Verder is er maar één publieke property — de ScreenState — en één publieke methode — de OnAction methode. Meer hierover in het Action subsectie.

Het basis ViewModel classe heeft twee eigenschappen die nodig zijn voor hun verwachtte werking. De eerste hiervan is dat ze niet composable zijn; hierdoor moet er geen *remember* toegevoegd worden zodat ze heen de lifecycle van de applicatie blijven. Dit werkt samen met de tweede eigenschap: ze volgen het singleton patroon. Dit garandeert dat elke nieuwe aanroeping van een VM identiek hetzelfde object zal geven. Aan de hand van deze twee eigenschappen kunnen VMs doorheen de levenscyclus van de applicatie geldig blijven.

### 3.1.2 Screen

Screens vormen de omgeving waarmee de gebruiker kan interageren. Dit zijn de “views” van in het MVC patroon. Het toont de gegevens op één manier ongeacht welke gegevens dit precies zijn. Al wat de Screen weet is dat er een dataset gegeven wordt in een zeker formaat en dat er een zeker stuk code uitgevoerd moet worden bij het klikken op de knop. Nogmaals, dit is een manier om de taken binnen de applicatie te splitsen. Wat het klikken op een knop doet maakt de Screen volledig niets uit. Of dit een externe library ophaalt, data schrijft naar een bestand of een Toast aanmaakt is irrelevant. Alles wat een component hoort te weten is dat er een functie bestaat bij het klikken van elke knop.

Aan de hand van de huidige state parameter wordt de correcte display vooruit gebracht. Elke screen heeft één publieke constructor en verschillende privé constructoren die voor de verschillende scenario’s dienen. Zo zijn er standard een “Success(value)”, “Failure(error)” en “Loading()”, drie vaak voorkomende scenarios, voornamelijk wanneer er data opgehaald moet worden.

```kotlin
@Composable
fun AccountScreen(
    state : AccountStateList,
    onAction : (AccountAction)->Unit,
    onLogOutClick : ()->Unit,
){
    when(state) {
        is AccountStateList.Failure ->
            AccountFailure(
                error = state.error,
                onRetry = { onAction(AccountAction.OnRetry) }
            )
        AccountStateList.Loading ->
            AccountLoading(
                onRetry = { onAction(AccountAction.OnRetry) }
            )
        is AccountStateList.Success -> {
            AccountSuccess(
                onAction = onAction,
                user = state.user,
            )
        }
        is AccountStateList.LogOut -> {
            onLogOutClick()
        }
    }
}
```

### 3.1.3 State

State worden gebruikt om door te geven aan de UI wat te tonen. Alle nodige informatie voor de screens wordt zo gegeven. Twee formaten worden doorheen de app gebruikt, volgens wat de context nodig heeft.

#### State List

State List data classes maken gebruik van Kotlin's sterke enum types en smart casting om verschillende mogelijkheden efficiënt te behandelen, zonder extra gevallen. Elke entry is een data class/object en heeft andere parameters. Zo weet de Screen dat een Failure entry een foutmeldingsstring bevat, terwijl een Success entry bvb. een CatProfile heeft. De publieke constructor van de screens krijgen een state als parameter vanuit het viewmodel en lossen intern elke entry op met een aparte composable functie.

Stel er wordt externe data opgehaald door het viewmodel, en er loopt iets mis. Het viewmodel kan een Failure state inzetten met een foutmelding. Dit belandt bij de Screen, waar een FailureScreen wordt gemaakt met de geleverde fout, die in een textvak wordt gezet. Stel de gebruiker herlaad de pagina, maar deze keer komt de data wel correct binnen. Het viewmodel zet dan de state op Success met de State Container als parameter.

```
sealed interface AccountStateList {
    data class Success(val user : CatProfile) : AccountStateList
    data object Loading : AccountStateList
    data class Failure(val error : String) : AccountStateList
    data object LogOut : AccountStateList
}
```

**State Container**

State Containers zijn, kort door de bocht, data classes waarmee alle data voor de UI in een bundle wordt verzameld. Zo moet er maar één parameter gegeven worden indien de UI er meerdere nodig heeft. Uiteraard bestaat er zo één per pagina; ze hebben niet de zelfde informatie nodig!

### 3.1.4 Action

Terwijl State gebruikt wordt om van het viewmodel naar de UI te communiceren, Action is net het omgekeerde. Zoals State List, Actions zijn enums (sealed classes), waardoor er makkelijk en veilig verschillende opties gemaakt kunnen worden. Met één centrale methode in het viewmodel kunnen alle nodige acties uitgevoerd worden zonder een nieuwe parameter aan te maken voor elke interactie. Zo wordt de code in de App file wat beperkt; het is anders makkelijk om honderden regels code te maken om alle interacties in te stellen voor elke pagina. In het viewmodel bestaat maar één publieke methode: *DoAction*, die als enige parameter een Action object van de overeenkomende type nodig heeft. Intern worden alle scenarios behandeld en het resultaat via het de State weergegeven in de UI.

Het beste voorbeeld hiervan is de Register pagina. Hier moet een emailadress en paswoord ingevuld worden, twee cruciale elementen van de gebruiker. Uiteraard mag het paswoord niet te kort of simpel zijn. Abstractie zegt dat de UI deze logika niet zelf mag uitvoeren — wat ook erg logisch is volgens mij — dus deze informatie moet tot het viewmodel geraken. Aan de hand van het DoAction methode van het viewmodel, doorgegeven via de constructor van de pagina, kan OnPasswordUpdate uitgevoerd worden, met het paswoord als invoer. Deze controleert of het voldoet en update de *canSignUp* property van de State Container.

## 3.2 Database

### 3.2.1 Optimisatie

Data ophalen en opslaan is één van de kenmerken van hedendaagse applicaties. Al kunnen we ervan uitgaan dat hardware over het algemeen steeds sterker en communicatie steeds sneller verloopt, moeten we rekening houden met sommige eigenschappen van mobiele telefoons. Deze hebben beperkte batterij en vaak maken gebruik van mobiele data i.p.v. wifi. Vaak of veel data ophalen neemt extra rekenkracht en verbruikt onnodig veel data, die bij sommige gebruikers beperkt kan zijn. Het is dus in het belang van de gebruikers om alle communicatie zo efficiënt en vlot mogelijk te maken om zo weinig mogelijk batterij en mobiele data te gebruiken.

Het zou volledig mogelijk zijn om dit project op te bouwen envan uitgaand dat iedere gebruiker over onbeperkte data en de nieuwste generatie telefoon beschikt. Dit zou mij veel moeite besparen op vlak van programmeer- en denkwerk, maar... daar ben ik zelf niet tevreden mee. Hardware stijgt razendsnel in rekenkracht, programma's kunnen meer en meer resources in beslag nemen, en deze app zal realistisch nooit meer dan drie gebruikers hebben (dank je wel, Oma). Nochtands kan ik het niet over mijn hart krijgen om niet te optimiseren waar mogelijk, enerzijds omdat mijn telefoon zelf best achterhaald is, maar anderzijds omdat programmeren zo moet in mijn visie van alles.

De eerste stap om dit probleem te benaderen is identificeren wanneer er nieuwe data moet opgehaald worden en deze momenten zo streng mogelijk reglementeren.

### 3.2.2 Structuur

## 3.3 Navigatie

# 4 Besluit