HO
GENT

Faculteit Bedrijf en Organisatie

Android Native Development in Kotlin versus het Flutter Framework, een vergelijkende studie

Navaron Bracke

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Johan Van Schoor
Co-promotor:
Kenneth Saey

Instelling: Endare BVBA

Academiejaar: 2019-2020

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Android Native Development in Kotlin versus het Flutter Framework, een vergelijkende studie

Navaron Bracke

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Johan Van Schoor
Co-promotor:
Kenneth Saey

Instelling: Endare BVBA

Academiejaar: 2019-2020

Tweede examenperiode

Woord vooraf

Ik zou van dit voorwoord gebruik willen maken om een aantal mensen te bedanken. Graag zou ik mijn co-promotor Kenneth Saey willen bedanken voor de hulp en feedback die hij mij heeft aangeboden. Ook zou ik mijn promotor Johan Van Schoor, lector aan de faculteit Bedrijf en Organisatie te Hogeschool Gent willen bedanken voor zijn steun tijdens de uitwerking van deze thesis. Ten slotte zou ik Jelle Geers en Jens De Wulf willen bedanken, voor hun feedback inzake allerhande organisatorische zaken.

Samenvatting

Flutter is een nieuw framework voor het ontwikkelen van mobiele applicaties. Er bestaat echter nog geen diepgaande analyse van de werking van het framework. Een dergelijke analyse is noodzakelijk om Flutter mee in rekening te kunnen nemen tijdens de keuze voor een Cross-Platform framework. Dit paper probeert dit hiaat te dichten, door middel van een vergelijkende studie tussen het Flutter framework en de native Android stack. De stack van het Android platform biedt een zicht op de features die de ontwikkelaars van mobiele apps verwachten van de te gebruiken software toolkit. Om tot de resultaten van de vergelijkende studie te komen, is het onderzoek vertrokken vanuit twee geïmplementeerde applicaties, één per onderzocht systeem. De uitgeschreven applicaties volgen een template dat garandeert dat de criteria die dit onderzoek behandeld aan bod komen. De gekozen template bevat criteria voor het uitwerken van user interfaces, het werken met asynchrone bewerkingen, het gebruiken van een lokale database, het opzetten van navigatie binnen een app, toepassen van vertalingen, het gebruik van applicatie permissies en het uitschrijven van software testen. De template voorziet ook twee performantie gerelateerde criteria voor het onderzoek, applicatiegrootte en de effectieve opstarttijd van een applicatie. Het laatste hoofdstuk van dit paper brengt alle bevindingen samen, om tot de conclusie te komen dat het framework al veel zaken aanbiedt én verbeterd ten opzichte van native Android. De jonge aard van het framework brengt wel met zich mee dat een aantal features niet efficiënter zijn in vergelijking met de equivalente werkwijze van de native Android toolkit. De bekomen resultaten creëren ook een aantal nieuwe mogelijkheden voor toekomstig onderzoek. De vergelijking met de toolkit voor native IOS applicaties ontbreekt nog. Ten tweede gaan de onderzoeksresultaten niet dieper in op de uiteindelijke ontwikkelingstijd van de uitgeschreven applicaties. Dit is een interessante metric om te onderzoeken maar valt jammer genoeg buiten de reikwijdte van dit onderzoek.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

Het aantal mobiele applicaties, apps voor 'mobiele' toestellen zoals een smartphone of tablet, neemt alsmaar toe. Dit is te verklaren door het feit dat er alsmaar meer van dit soort hardware op de markt te vinden is. Dit brengt met zich mee dat het ontwikkelen van zulke applicaties aantrekkelijker wordt voor softwarebedrijven. Deze kunnen hiermee namelijk veel meer klanten bereiken. Op dit moment is het zo dat elk softwarebedrijf dat zulke applicaties maakt, moet beslissen welke manier van ontwikkelen het beste geschikt is. Toen het ontwikkelen van mobiele applicaties nog maar net aan het opkomen was, was er maar één geschikte optie om een implementatie van een mobiele applicatie uit te werken. Deze manier van werken heet native development en geeft ontwikkelaars van mobiele applicaties de mogelijkheid om rechtstreeks met het native systeem te werken. Deze werkwijze heeft wel een groot nadeel, de geschreven code kan enkel werken op het platform van het gekozen native systeem. Naast Native Development is er nog een tweede manier om mobiele applicaties te schrijven. De dag van vandaag zijn er verschillende doelplatformen. Het uitwerken van een project per doelplatform met behulp van native development kan vrij duur uitvallen voor bedrijven. De Cross-Platform ontwikkelingstechniek probeert hiervoor een oplossing te bieden. Deze ontwikkelingstechniek maakt gebruik van een framework dat het mogelijk maakt om één softwareproject te gebruiken voor meerdere platformen. Om de keuze te maken tussen Native Development enerzijds en Cross-Platform Development anderzijds is het echter noodzakelijk dat de voor en nadelen van elke werkwijze goed in kaart kunnen worden gebracht. Hiervoor moeten de Cross-Platform frameworks, zoals React Native of Cordova, grondig worden vergeleken met een native werkwijze.

Sinds december 2018 is er echter een interessante nieuwe speler onder de Cross-Platform frameworks, Flutter. Flutter is een User Interface Toolkit, een softwarepakket om mobiele applicaties te ontwikkelen, ontworpen door Google. Omdat Flutter - in softwaretermen

- nog vrij jong is, brengt dit echter een probleem met zich mee: Indien ontwikkelaars moeten kiezen tussen Native Development of Cross-Platform Development, dan moeten de ontwikkelaars inzicht hebben in de werking van dit framework. Maar aangezien het framework nog zeer jong is, is het inschatten van dit framework niet zo evident.

1.1 Probleemstelling

Endare is een softwarebedrijf dat zich specialiseert in het ontwikkelen van mobiele applicaties. Voordat de uitwerking van een nieuw softwareproject kan starten, maken de ontwikkelaars de keuze tussen een native of Cross-Platform werkwijze. Hiervoor moeten de ontwikkelaars alle bestaande Cross-Platform frameworks kunnen inschatten. Omdat het Flutter framework nog nieuw is hebben de ontwikkelaars van Endare weinig tot geen ervaring met dit framework. Dit maakt deze beslissing des te moeilijker. Daarom is het noodzakelijk dat er een duidelijk beeld wordt geschetst over hoe het Flutter framework zich verhoudt tot native development. Enkel dan zullen de developers, van bedrijven zoals Endare, Flutter in acht kunnen nemen bij het maken van een beslissing rond het kiezen van een manier van ontwikkelen.

1.2 Onderzoeksvraag

Dit onderzoek zal een antwoord proberen te vinden op de volgende onderzoeksvragen:

Is het ontwikkelen van features met het Flutter framework even efficiënt als de werkwijze die gebruik maakt van een native ontwikkelingstechniek?

Het onderzoek zal proberen te achterhalen of het ontwikkelen van features voor een applicatie met behulp van het Flutter framework, efficiënt kan gebeuren. Zijn er extra stappen nodig om een bepaalde feature te ontwikkelen, die niet nodig zijn bij native ontwikkeling? Of is het net de native ontwikkeling die extra stappen nodig heeft bij de uitwerking van feature X? Bijkomend zal het onderzoek kijken naar het gewicht van de te ondernemen stappen. Een feature met één uit te werken stap is niet noodzakelijk efficiënter, deze stap kan zwaarder uitvallen ten opzichte van een aantal kleinere stappen in het andere systeem.

Is het Flutter framework, op vlak van features, even toereikend als een native ontwikkelingstechniek?

Het onderzoek zal ook proberen achterhalen of de huidige versie van het Flutter framework[1] een equivalente implementatie mogelijk maakt voor elk van de features uit de native development stack van het Android platform. Met andere woorden, kan een ontwikkelaar (technisch gezien) een native applicatie volledig herschrijven in Flutter, zonder te moeten inboeten op functionaliteit.

[1] De *stable* versie bij de start van de uitwerking van deze thesis is versie 1.12.

1.3 Onderzoeksdoelstelling

Dit onderzoek is een vergelijkende studie tussen het Flutter framework en native ontwikkeling in Android. Dit onderzoek zal een beeld proberen schetsen over het werken met het Flutter framework. Om dit beeld iets tastbaarder in kaart te brengen zal Flutter worden vergeleken met een native development workflow in het Android ecosysteem.

Om tot het eindresultaat te komen, zal het onderzoek gebruik maken van twee uitgeschreven applicaties. De eerste applicatie zal een native Android implementatie voorzien, met behulp van de Kotlin programmeertaal. De tweede applicatie zal een Flutter implementatie voorzien, met behulp van de Dart programmeertaal. Deze applicaties zullen elk een algemeen en vooraf bepaald concept volgen. Dit concept zal er voor zorgen dat elk van de evaluatie criteria van dit onderzoek aan bod komen.

De evaluatie criteria waarvoor het concept de aanwezigheid zal garanderen zijn:

- Internationalisering
- Navigatie
- Persisteren van gegevens
- User Interfaces
- Asynchroon Werk
- Permissies
- Software Testing
- Opstarttijd van de applicatie
- Grootte van de applicatie

Het onderzoek zal vertrekken vanuit de volgende hypotheses:

Het Flutter framework kan niet voor alle onderzoeks-criteria een volwaardige optie bieden, in tegenstelling tot Android.

Het Flutter framework is niet de efficiëntste van de twee kandidaten in ten minste de helft van de onderzoeks-criteria.

Het beoogde resultaat van dit onderzoek is het beantwoorden van de opgestelde onderzoeksvragen, met een doorslaggevend antwoord. Hiervoor zal er beroep worden gedaan op de bovenstaande hypotheses en de evaluatie criteria.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

Hoofdstuk 2 licht de huidige research naar het Flutter framework toe.

Hoofdstuk 3 zal de methodologie en de gebruikte onderzoekstechnieken van dit onderzoek bespreken.

Hoofdstuk 4 gaat dieper in op de uitwerking van user interfaces voor de twee behandelde systemen.

Hoofdstuk 5 zal het werken met asynchrone bewerkingen bekijken.

Hoofdstuk 6 bekijkt een aantal mogelijkheden voor de uitwerking van een lokaal persistentie-systeem, door middel van lokale databases.

Hoofdstuk 7 bespreekt hoe beide applicaties kunnen gebruik maken van navigatie.

Hoofdstuk 8 zal bekijken hoe beide systemen omgaan met het vertaalbaar maken van een applicatie.

Hoofdstuk 9 zal verder ingaan op applicatie permissies.

Hoofdstuk 10 bespreekt hoe beide systemen het mogelijk maken om software testen uit te werken.

Hoofdstuk 11 zal de twee systemen met elkaar vergelijken op vlak van opstart-performantie.

Hoofdstuk 12 zal bepalen welk systeem de kleinste bestandsgrootte kan bieden voor een uitvoerbaar applicatiebestand.

Hoofdstuk 13 zal het idee rond het oproepen van native code via Method Channels uitleggen.

Hoofdstuk 14 zal een conclusie formuleren op de onderzoeksvragen van dit onderzoek. Dit hoofdstuk zal naast het formuleren van de conclusie ook een aanzet geven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

2.1 Inleiding van de literatuurstudie

Dit hoofdstuk vat de bestaande research rond het Flutter framework samen. Deze samenvatting biedt ook een inzicht in de tekortkomingen van de bestaande research.

2.2 Reeds bestaand onderzoek

Het onderzoek van Tuusjärvi (Tuusjärvi, 2019) probeert via een Proof Of Concept te achterhalen welk Cross-Platform framework geschikt is om mobiele applicaties te ontwikkelen. De onderzoekers vergeleken hiervoor Flutter, React Native en Xamarin Forms. Dit onderzoek kwam tot de conclusie dat Flutter de beste kandidaat was voor de uitwerking van de gebruikte applicatie. De onderzoekers wezen hiervoor naar de voordelen van de *Hot Reload* functionaliteit van het Flutter framework. Hot Reload injecteert stukken gewijzigde code in de draaiende applicatie, wat een snellere ontwikkeling als resultaat heeft. De onderzoekers zagen de aanwezigheid van Ahead-Of-Time compilatie als een succesfactor voor het Flutter framework. Ten slotte concludeerden de onderzoekers dat de Dart programmeertaal snel aan te leren is omwille van de gelijkenis met andere talen. De beperkte reikwijdte van dit onderzoek bracht wel met zich mee dat het gedeelte rond user interfaces niet uitvoerig aan bod kwam.

Het onderzoek van Yatsenko (Yatsenko, Obodiak & Yatsenko, 2019) vergelijkt een aantal Cross-Platform frameworks met elkaar. De onderzoekers hebben ervoor gekozen om Flutter, React Native, Cordova en Xamarin met elkaar te vergelijken. De onderzoekers keken kort naar de samenhang van de verschillende componenten van elk framework. De

conclusie van dit onderzoek wijst React Native en Flutter aan als de twee voornaamste frameworks. De onderzoekers wezen hiervoor naar de populariteit van deze frameworks en het feit dat de andere frameworks niet volledig aan de requirements van mobiele applicatieontwikkeling voldoen. Dit onderzoek vermeld ook de goede documentatie die het Flutter framework aanbiedt aan ontwikkelaars. De onderzoekers van dit onderzoek blijven op een high-level niveau hangen en gaan niet verder in op de implementatiedetails.

Rodríguez-Sánchez Guerra vergelijkt het Flutter framework met drie andere Cross-Platform frameworks, React Native, Weex en Ionic. (Rodríguez-Sánchez Guerra, 2018) Dit onderzoek probeert het belang van Cross-Platform development meer op de kaart te zetten. De onderzoeker koos ervoor om de code kwaliteit en uitvoeringstijd van de verschillende frameworks te vergelijken. Uit dit onderzoek is gebleken dat er nog geen vergelijking was tussen een dergelijk Cross-Platform framework en applicaties die gebruik maken van een native ontwikkelings techniek. De onderzoeker vermeld dit laatste als een mogelijk idee voor toekomstig onderzoek.

Het onderzoek van Dang en Skelton (Dang & Skelton, 2016) kijkt naar het Flutter framework in een meer educatieve context. De researchers van dit onderzoek staan zelf in het onderwijs. Het leek voor de onderzoekers interessant om te onderzoeken of het Flutter framework een geschikte kandidaat is voor de lessen binnen het curriculum. Hiervoor vertrekt het onderzoek vanuit een vergelijking tussen een aantal Cross-Platforms zoals React Native, Xamarin Forms en Flutter. Dit onderzoek bekijkt een aantal belangrijke criteria zoals user interfaces, animatie en threading. Het onderzoek komt tot de conclusie dat het Flutter framework de betere keuze is voor het onderwerp van een les. Dit onderzoek vertrok wel van een oudere versie van het Flutter framework, het moment van publicatie ligt vóór de datum van versie 1.0 van het Flutter framework.

Het onderzoek van Fayzullaev (Fayzullaev, 2018) vergeleek het Flutter framework met twee minder bekende cross-platform frameworks, Kotlin Native (een cross-platform framework dat gebruik maakt van de Kotlin programmeertaal) en Multi-OS Engine. Dit onderzoek was vooral een exploratieve studie en ging niet diep in op de werking van de frameworks.

Het onderzoek van Dagne (Dagne, 2019) geeft een kort overzicht van het Flutter framework. Het onderzoek illustreert het idee rond de Widgets van Flutter. Vervolgens bespreekt de onderzoeker hoe stijlen en themas het mogelijk maken om een applicatie te voorzien van kleuren uit een kleurenpallet. Het onderzoek sluit af met een voorbeeld van een Method Channel. Het onderzoek vermeld ook het snelle en vlotte ontwikkelingsproces van het framework. Jammer genoeg gaat dit onderzoek niet verder in op andere belangrijke componenten van het framework.

Het onderzoek van Gonsalves (Gonsalves, 2019) vergelijkt Flutter met een Cordova applicatie, een native IOS applicatie en een native Android app. Dit onderzoek kijkt naar de algemene ervaring tijdens de ontwikkeling, de kwaliteit van de uitgeschreven applicaties, toegang tot native hardware en een aantal performantie gerelateerde criteria zoals geheugengebruik en de tijd die nodig is voor een overgang tussen twee schermen. Voor de grootte van de Android applicatie keek de onderzoeker naar de *debug* versie van de appli-

catie. Dit is een duidelijke fout van de onderzoeker, aangezien deze niet geminimaliseerd is. Het onderzoek meet ook de hoeveelheid RAM geheugen dat de applicaties gebruiken. De onderzoeker maakt hier de fout van te weinig te testen, per systeem heeft de onderzoeker drie metingen uitgevoerd. De onderzoeker moet minstens 30 metingen nemen om een goede steekproef uit te kunnen voeren.

Het onderzoek van Wu (Wu, 2018) vergelijkt het Flutter framework met React Native. Het onderzoek bekijkt hoe beide frameworks navigatie binnen een applicatie mogelijk maken. Vervolgens bekeek de onderzoeker hoe de twee frameworks omgaan met state management. De onderzoeker keek ook naar de ontwikkeling van user interfaces in beide systemen. De onderzoeker vergeleek de frameworks ook op vlak van performantie. Hiervoor heeft de onderzoeker de performantie van het scrollen binnen een lijst onderzocht. De onderzoeker heeft ook gekeken naar de performantie van bestandsoperaties. Dit omvat het lezen en schrijven naar een bestand. De onderzoeker concludeerde dat beide technologieën goede keuzes zijn voor Cross-Platform developers. Het gedeelte rond user interfaces van dit onderzoek, ging echter niet diep genoeg in op een aantal belangrijke concepten van het Flutter framework.

Sharma en Gupta vergelijken het Flutter framework met React Native. (Sharma & Gupta, 2020) Dit onderzoek keek vooral naar de tooling die ontwikkelaars ondersteunt tijdens het ontwikkelen van software. Het onderzoek bespreekt of de frameworks tools aanbieden om software testen uit te werken. Hieruit blijkt dat React Native bij de user interface testen tekort schiet. Flutter kan ontwikkelaars voor dit soort testen wel de nodige tools aanbieden. Hoe deze tools werken, hebben de onderzoekers niet besproken. De onderzoekers namen ook de setup van een nieuw project onder de loep. React Native blijkt slechte documentatie te hebben rond het opzetten van een Android project. De onderzoekers merkten op dat de setup van het Flutter project gemakkelijker was dan de setup van het React Native project. De onderzoekers besloten uiteindelijk dat de keuze voor een Cross-Platform framework iets is waar ontwikkelaars zelf over moeten beslissen.

2.3 Nood aan deze bachelorproef

Uit de reeds bestaande research valt vooral te concluderen dat er vrij weinig onderzoek is naar het Flutter framework. Dit valt waarschijnlijk te verklaren door de leeftijd van het framework. Wat er ook opvallend is, is het feit dat veel van de bestaande research vertrekt vanuit een zogenaamde 'framework vergelijking'. In dit soort vergelijkingen nemen de researchers twee Cross-Platform frameworks onder de loep, om ze dan met elkaar te kunnen vergelijken. Er is echter weinig research die een totaalbeeld kan geven over hoe het framework in elkaar zit, de bestaande vergelijkingen zijn nog te oppervlakkig. Ook is er weinig te vinden over de impact van het overschakelen van een Native Development workflow naar een Cross-Platform workflow. Het ontbreken van dit laatste is jammer genoeg een hiaat in het huidige onderzoek rond Cross-Platform frameworks.

3. Methodologie

3.1 Inleiding Methodologie

Om een goed beeld te kunnen vormen van het Flutter framework, maakt dit onderzoek gebruik van een experiment. Dit hoofdstuk zal de structuur van het gekozen experiment uitleggen.

3.2 Het Experiment

Dit onderzoek maakt gebruik van twee testapplicaties. De eerste applicatie is een Android app, deze vertegenwoordigt de Native development workflow. De tweede applicatie is een Flutter app. Deze applicatie vertegenwoordigt de Cross-Platform framework workflow. Het onderzoek maakt tijdens de ontwikkeling van beide applicaties gebruik van de Android Studio Integrated Development Environment. Beide applicaties maken gebruik van de Software Development Kit van Android. Android Studio maakt van deze SDL gebruik om de Android gerelateerde tooling te voorzien. De native Android applicatie maakt gebruik van de Kotlin programmeertaal. De Flutter applicatie maakt gebruik van de Dart programmeertaal. Tijdens de ontwikkeling van de Flutter applicatie bieden de Flutter en Dart plugins[1] ondersteuning bij de uitwerking. De ontwikkelingsfase zal van deze plugins gebruik maken.

[1] Dit zijn plugins voor Android Studio.

3.3 De Android App

De Android applicatie voorziet de gebruiker van een startscherm, zoals te zien in figuur 3.1. Op dit startscherm bevinden zich twee tabbladen. Het eerste tabblad biedt een showcase aan. Deze showcase geeft de gebruiker de mogelijkheid om een aantal zaken uit te proberen. Het tweede tabblad biedt de gebruiker de mogelijkheid om de achterliggende datastore die de applicatie gebruikt, te veranderen. De verschillende datastores die de gebruiker kan kiezen, staan in voor de persistentielaag van de applicatie. Deze maken het mogelijk om gegevens op te slaan op het opslagmedium van het toestel. Het selecteren van deze datastore is een puur illustratief voorbeeld om aan te tonen dat de verschillende datastores werken.

De eerste optie uit de showcase geeft de gebruiker een paneel om te experimenteren met de werking van asynchrone bewerkingen. Zo kan de gebruiker om een of meerdere waarden vragen met behulp van een knop. Ook kan de gebruiker vragen om een foutmelding weer te geven, deze foutmelding stelt een gefaalde bewerking voor.

De tweede optie die de gebruiker kan kiezen, visualiseert het concept van navigatie binnen een mobiele applicatie. De gebruiker kan hier experimenteren met verschillende vormen van navigatie. De gebruiker kan kiezen tussen een niet-geneste navigatie en een geneste navigatie. Beide overgangen zullen gepaard gaan met een animatie. De gebruiker krijgt ook de mogelijkheid om een argument door te geven tijdens een navigatie tussen schermen.

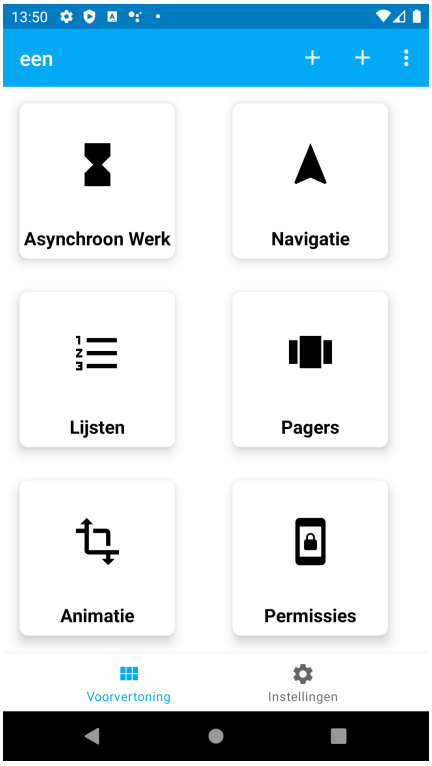
De derde en vierde optie zijn verwant aan elkaar. Hier zal de gebruiker kunnen kiezen tussen twee componenten die elk instaan voor het tonen van een (groot) aantal data elementen.

De vijfde optie binnen de showcase geeft de gebruiker een overzicht van een aantal mogelijke animaties die een Android applicatie zou kunnen gebruiken.

De zesde optie geeft de gebruiker de mogelijkheid om te werken met applicatiepermissies. De gebruiker kan door middel van een aantal knoppen op dit scherm, om permissies vragen. Indien de gebruiker een bepaalde permissie niet heeft goedgekeurd, dan zal de applicatie voor deze permissie een melding tonen.

Ten slotte zal de gebruiker kunnen kiezen voor een optie die het gebruik van formulieren en invoer illustreert. De gebruiker kan op dit scherm experimenteren met een aantal verschillende vormen van invoer, van een versleepbare slider tot een invoerveld voor tekst. De gebruiker krijgt ook de kans om deze invoer op te slaan in de datastore die via het tweede tabblad opgegeven wordt.

Het keuzescherm en het formulier van de applicatie zal zich aanpassen aan de schermafmetingen en oriëntatie.



Figuur 3.1: Het startscherm van de applicatie die zal worden gebruikt als voorbeeld. (Bracke, 2020)

3.4 De Flutter App

De Flutter applicatie volgt hetzelfde idee als de Android app. De functionaliteit van beide applicaties zal nagenoeg identiek zijn, tenzij details van de uiteindelijke implementatie dit verhinderen.

De implementatie van de Flutter app maakt gebruik van de Widgets uit de Material Design Widget Catalogue, deze volgen namelijk de stijlgids van Android. Een aantal Widgets krijgen een equivalent voorbeeld uit de Cupertino Catalogue. De Cupertino Widget Catalogue bevat Widgets die de stijlgids van het IOS platform volgen. Het doel van de extensie binnen deze implementatie, is enkel het demonstreren van het aanpassen van de user interface naar een ander doelplatform, zijnde IOS in dit geval. Dankzij deze extensie kan dit onderzoek het 'cross-platform' gedeelte van het framework behandelen.

Ten slotte voorziet het onderzoek een appendix die het idee rond Method Channels uitlegt. Method Channels zijn de manier die het Flutter framework aanbiedt om native code aan te spreken.

3.5 Toepassen van de onderzoekscriteria

De hoofdstukken die volgen zullen elk een deel van de onderzoekscriteria behandelen.

Het hoofdstuk Asynchroon Werk vergelijkt beide applicaties op vlak van asynchrone bewerkingen. Het aantal lijnen code van een geïmplementeerde asynchrone bewerking zal dienen als metric voor de resultaten.

Het hoofdstuk Navigatie zal de implementaties van de applicaties vergelijken aan de hand van de verschillende componenten die nodig zijn om een basis navigatie op te zetten. De vergelijking telt het aantal componenten én de lijnen code per component van de twee geïmplementeerde systemen.

Het hoofdstuk Persistentie spitst zich toe op een aantal verschillende mogelijkheden die beide systemen beschikbaar stellen om lokale persistentie op te zetten. Dit houdt in dat een lokale database, die zich binnen de applicatie bevindt, instaat voor het persistent maken van gegevens. Het hoofdstuk voorziet voor elke database een korte uitleg over de beschikbare features, wanneer de database kan gebruikt worden en behandeld de voor- en nadelen van elke oplossing. Indien een database een implementatie heeft voor native Android én Flutter, zal deze onderworpen worden aan een metric die het aantal lijnen code telt voor een mogelijke implementatie.

Het hoofdstuk User Interfaces gaat dieper in op het gebruik van user interfaces in native Android en Flutter. Ten eerste zal het hoofdstuk *state management* behandelen. Vervolgens komen een aantal Flutter specifieke concepten aan bod, zoals sleutels en *InheritedWidget*s. Hierna schakelt het hoofdstuk over naar het vergelijken van een aantal user interface componenten die native Android aanbiedt, met de overeenkomstige Widgets van Flutter. Deze vergelijking zal nagaan of de Widgets van Flutter minder werk vragen in vergelijking met de overeenkomstige componenten van native Android. Hiervoor zal de vergelijking het aantal lijnen code bekijken. Naast het evalueren van de user interface elementen, zal het hoofdstuk de animatiesystemen vergelijken op vlak van gebruiksvriendelijkheid en het aantal lijnen code om een animatie op te zetten. Ten slotte zal dit hoofdstuk nagaan in welk van de twee systemen de beste aanpak kan bieden om responsive designs te maken. Deze laatste vergelijking zal nagaan of de implementaties die Flutter voorziet om layouts aan te passen voor verschillende schermdimensies en oriëntaties de bestaande implementatie van de native Android app kan evenaren in het aantal geschreven lijnen code en gebruiksgemak.

Het hoofdstuk Internationalisering gaat dieper in op het toevoegen van vertalingen. Dit hoofdstuk zal nagaan hoe moeilijk het is om een vertaling toe te voegen voor een nieuwe taal. Concreet zal de vergelijking proberen te achterhalen hoe gemakkelijk een applicatie die reeds in het Nederlands beschikbaar is, ook ondersteuning kan bieden voor het Engels.

Het hoofdstuk Permissies bespreekt het gebruik van applicatiepermissies in de twee applicaties. Concreet vergelijkt dit hoofdstuk de werkwijzen om een permissie te vragen aan een gebruiker. De vergelijking zal het aantal lijnen code tellen voor de implementaties van de twee systemen.

Het hoofdstuk Software Testing spitst zich toe op de mogelijkheden om software testen te schrijven. Hiervoor krijgen de twee applicaties voor elk type software test die dit onderzoek behandeld, een implementatie. Dit onderzoek zal Unit Testen, User Interface Testen

en Integration Testen behandelen. Elke test krijgt een evaluatie op basis van het aantal lijnen code per implementatie. Naast het uitwerken van deze testen, kijkt dit hoofdstuk ook naar de moeite die een ontwikkelaar moet doen om een testbare applicatie uit te werken.

Voor het meten van de grootte en de opstarttijd van de applicatie, maakt het onderzoek gebruik van een aparte applicatie voor beide systemen. Deze applicatie volgt het klassieke 'Hello World' voorbeeld. Dankzij deze template blijven de resultaten representatief en blijven de metingen vrij van externe factoren. De meting voor de applicatiegrootte vertrekt vanaf een *release* build met *minifyEnabled* ingeschakeld. Dit is een build configuratie die de applicatie minimaliseert en waar er geen ontwikkelingstools zoals een debugger aanwezig zijn.

De Android versie van de Hello World applicatie zal voor de metingen van de opstarttijd de profiling tools van de Android SDK gebruiken. De opstarttijd van de Flutter versie van Hello World maakt gebruik van de *profile* modus. Dit is een speciale build variant voor Flutter apps die performantie gerelateerde testen mogelijk maakt. De metingen voor de opstarttijd maken gebruik van de fysieke telefoon om de resultaten te bekomen.

De file explorer van de computer van de onderzoeker geeft de resultaten terug voor de uiteindelijke grootte van de applicaties.

3.6 Gebruikte Hardware

Voor het bekomen van de resultaten van dit onderzoek, maakt het onderzoek gebruik van een telefoon als hardware.

Deze telefoon heeft de volgende specificaties:

- Fabrikant: Motorola
- Model: motorola one
- CPU: Qualcomm Snapdragon 625
- OS: Android Q
- RAM geheugen: 4 gigabyte
- ROM geheugen: 64 gigabyte
- Schermresolutie: 1520 x 720 pixels

4. Gebruikers Interface

4.1 Inleiding

Een user interface is vrijwel het belangrijkste deel van een mobiele applicatie. Het is immers het enige wat de gebruiker ziet en kan gebruiken. Dit hoofdstuk zal uitleggen wat een user interface precies is en hoe een user interface te implementeren is.

4.2 Wat is een user interface

Vooraleer de verschillende aspecten van een user interface aan bod komen, is het noodzakelijk om te definiëren wat een user interface inhoudt. Een user interface is het middel waarmee een gebruiker kan communiceren met een applicatie. Mobiele applicaties bevatten een Graphical User Interface (GUI). De GUI presenteert de grafische elementen van de applicatie aan de gebruiker, door middel van het tekenen van elementen op een fysiek scherm. Indien de gebruiker via deze elementen een interactie uitvoert, zal de GUI deze interactie opvangen en doorgeven aan de rest van de applicatie.

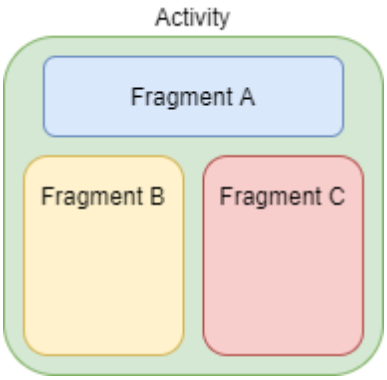
4.3 Opstellen van een user interface

4.3.1 User Interfaces in Android

Een Android applicatie opstarten kan via de Application klasse. Dit is een object dat de 'applicatie' voorstelt. Dit object heeft een levenscyclus, de 'Lifecycle'. Een applicatie

kan zich namelijk in een aantal situaties bevinden, zoals net gecreëerd zijn of zich in een actieve staat bevinden. In elk van deze situaties zal de applicatie communiceren met zijn levenscyclus en de onderliggende componenten. De applicatieklasse heeft zelf echter geen visuele componenten.

Voor het visuele aspect van Android apps zijn er twee componenten, Activities en Fragments.



Figuur 4.1: Een Activity omvat het hele scherm. Binnen een Activity kunnen meerdere Fragments bestaan. (Bracke, 2020)

Een Activity is een component die conceptueel gezien een 'scherm' voorstelt. De visuele component van deze Activity is een View. Een Activity bevat naast een View ook een Lifecycle. Dit is de 'levenscyclus' van de Activity en deze staat in verbinding met de levenscyclus van de applicatie. Views krijgen een layout definitie van een xml-bestand. Het *onCreate* Lifecycle event maakt deze layouts aan door de definities als het ware op te blazen. De definitie van een View bestaat uit een boomstructuur die andere View-elementen bevat.

Codevoorbeeld 4.1: Layouts definiëren in native Android met XML bestanden. (Bracke, 2020)

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.appcompat.widget.AppCompatTextView
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/app_name"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Het gebruik van Activities leidt echter tot een probleem, per scherm is er nood aan een Activity. Om dit probleem op te lossen bestaat er een andere component, de Fragment. Een Fragment is een modulaire user interface component en maakt het mogelijk om de Activity op een meer modulaire manier van een user interface te voorzien. Net zoals Activities hebben Fragments een Lifecycle. De lifecycle van een Fragment en een Activity zijn grotendeels gelijk, op een paar verschillen na.

Net zoals Activities, hebben Fragments ook een View. Het aanmaken van de layout van deze View gebeurt in het *OnCreateView* lifecycle event.

Met behulp van verschillende Fragments is het opbouwen van een modulaire user interface voor een Activity toch mogelijk. Een voorbeeld van een dergelijke compositie is te zien in figuur 4.1.

Android Jetpack - De Compose toolkit

Het opbouwen van de layouts via de xml-bestanden is lang de enige manier geweest. Omdat dit een nogal statische manier van werken is, hebben de ontwikkelaars bij Google gezocht naar een eenvoudigere manier.

Android Jetpack, een library toolkit van Google waar een aantal grote componenten voor native Android development in verwerkt zitten, heeft sinds kort een nieuwe component. De Compose toolkit is een tool die developers een manier biedt om, met minder code, user interfaces op te bouwen. Compose maakt gebruik van een meer declaratieve programmeerstijl, in tegenstelling tot de statische xml-bestanden.

Compose biedt developers de mogelijkheid om via zogenaamde 'composable functions' user interfaces te bouwen. (Sproch, Powell & Guy, 2019)

Dit onderzoek zal echter niet verder ingaan op de Compose toolkit, aangezien deze nog in het Developer Preview stadium zit.

Codevoorbeeld 4.2: Een Composable Function definiëren met Android Compose. (Bracke, 2020)

```
class MainActivity : AppCompatActivity(){
    override fun onCreate(...){
        super.onCreate(...)
        setContent {
            HelloWorld("Android")
        }
    }
}

@Composable
fun HelloWorld(text: String){
    Text(text = "Hello $text!")
}
```

4.3.2 User Interfaces in Flutter

Aangezien Flutter moet kunnen samenwerken met het native Android systeem, maakt de Flutter applicatie een aantal aanpassingen. Flutter apps maken gebruik van een *FlutterApplication* en een *FlutterActivity*, dit zijn Flutter specifieke implementaties voor de Application en de Activity klassen die nodig zijn voor een Android applicatie.

Het Flutter framework pakt het idee rond user interfaces anders aan dan native Android. Flutter zal via de FlutterActivity een *FlutterEngine* in de applicatie steken. De Flutter Engine staat in voor het registreren van de gebruikte *Method Channels*, zij het zelfgeschreven of van gebruikte plugins. Hoofdstuk 13 gaat dieper in op de werking van deze Method Channels. De FlutterEngine is het contactpunt met de achterliggende Skia renderer. Skia tekent de user interface elementen van de Flutter applicatie, ook wel Widgets genoemd, op het scherm. In vergelijking met native Android komen de Widgets overeen met Fragments. Beide stellen een herbruikbaar user interface element voor.

4.4 State Management

Software applicaties moeten naast het presenteren van een user interface, ook in staat zijn om de gebruiker van bepaalde functionaliteit te kunnen voorzien. Een applicatie zonder enige tastbare functionaliteit, biedt namelijk weinig waarde aan een gebruiker. Daarom houden software applicaties een soort 'huidige status' bij. Hierin zit de informatie die betrekking heeft op situatie-afhankelijke zaken, zoals bijvoorbeeld een object dat werd geselecteerd uit een lijst van items, of de gegevens van een formulier. Applicaties houden echter niet alleen dit soort logica vast. Er bestaat ook informatie die meer gericht is naar de hardware waarop de applicatie draait. Een voorbeeld van dit soort informatie is de huidige oriëntatie van het scherm of de taal die de gebruiker heeft ingesteld. Deze laatste vorm van informatie heeft echter een bijwerking. Deze informatie kan, gedurende de uitvoering van een applicatie, veranderen. Een dergelijke verandering heet een configuratie verandering (Engels: Configuration Change). Bij het optreden van een configuratieverandering gaat de staat van de applicatie verloren. Om de applicatiestaat toch te behouden na een configuratieverandering hebben zowel native Android en Flutter oplossingen ter beschikking gesteld.

4.4.1 Android ViewModels

Een native Android applicatie kan ervoor zorgen dat de staat behouden blijft door deze te bewaren in een instantie van de *ViewModel* klasse. Dit is een speciale klasse die specifiek ontworpen is om de configuratieveranderingen te overleven. Naast ViewModels bestaan er binnen native Android nog andere mogelijkheden om de applicatiestaat te behouden, maar het gebruik van een ViewModel instantie is wel de meest voorkomende.

Naast de gewone ViewModel klasse, bestaat er nog een tweede implementatie, de *AndroidViewModel* klasse. Deze klasse breidt de ViewModel klasse uit met de toegang tot

het applicatie object. Deze klasse heeft toegang tot de applicatie resources, zoals taal afhankelijke tekst.

Om de ViewModels te verbinden met de user interface, maakt native Android gebruik van DataBinding (Sun, Chen & Yu, 2017). Dit is een manier om de veranderingen van de user interface te koppelen aan de applicatie logica. DataBinding genereert een klasse op basis van de layout bestanden voor een Activity of Fragment. Dit object maakt het mogelijk om de applicatielogica vast te hangen aan de layouts.

4.4.2 Stateful en Stateless Widgets

Om het probleem rond state management op te kunnen lossen, biedt het Flutter framework twee versies aan van de Widget klasse, Stateless en Stateful Widgets.

Stateless Widgets kunnen wel eigenschappen bevatten maar de referenties naar de waarden van hun eigenschappen mogen niet veranderen, omdat de Stateless Widgets zelf 'immutable' zijn. Een Stateless Widget is geschikt voor een user interface component die zelf geen staat moet vasthouden. Een simpel voorbeeld van een Stateless Widget is een knop die tijdens het aanklikken visuele feedback aan de gebruiker toont. Deze knop heeft enkel de taak om de rest van de applicatie te laten weten dat er op deze knop is geklikt en onderneemt zelf verder geen acties.

Stateful Widgets bevatten, in tegenstelling tot Stateless Widgets, wél een staat die kan veranderen gedurende de loop van de applicatie. Hiervoor maken de Stateful Widgets gebruik van een speciale klasse, de *State* klasse. De informatie die in deze klasse zit, blijft behouden tijdens configuratieveranderingen. De eigenschappen van de *Stateful Widget* blijven wel immutable, net zoals bij de Stateless Widgets. Een voorbeeld van een Stateful Widget is een knop die, tijdens het aanklikken, een verandering aanbrengt aan zijn tekst. De specifieke tekst van de knop zal in dit geval onderdeel zijn van de State klasse van deze Widget.

Om tijdens de loop van de applicatie de staat van een Stateful Widget aan te passen, biedt de State klasse de *setState()* methode aan. Dit is een methode die een callback, een uitvoerbaar stukje code, oproept en daarna aan de applicatie vraagt om de Widget opnieuw te tekenen op het scherm. Zo krijgt de user interface een update met de nieuwe gegevens.

4.4.3 Keys

Flutter biedt de Widget klasse een handige extra eigenschap in de vorm van de Key klasse. Keys, sleutels in het Nederlands, bieden een aantal extensies rond Widgets aan. Keys maken het mogelijk om het framework te laten weten dat twee Widgets van hetzelfde type, die op hetzelfde niveau in de boom zitten, toch opnieuw moeten worden getekend. Een sleutel kan ook de scroll positie van een lijst bewaren[1]. (Fortuna, 2019)

[1] De sleutel geeft toegang tot de *State* die de effectieve waarde bevat.

Dit onderzoek zal de twee meest voorkomende types van sleutels, de *ValueKey* en de *GlobalKey*, behandelen. Er bestaan nog andere types sleutels zoals de *ObjectKey* en de *UniqueKey* maar deze vallen buiten de reikwijdte van dit onderzoek.

ValueKey

Een ValueKey is een afgeleide van de algemene *Key* klasse. Dit soort sleutel gebruikt een logische waarde, zoals een stuk tekst of een geheel getal, om de sleutel uniek te maken.

GlobalKey

Net zoals de ValueKey is de GlobalKey een sleutel voor een Widget. GlobalKeys, letterlijk vertaald komt dit neer op 'globale sleutels', zijn een type sleutel die iets meer functionaliteit bieden dan de ValueKeys. Een GlobalKey biedt namelijk toegang tot de huidige instantie van de State klasse van een StatefulWidget.

Deze sleutel biedt de toegang tot methodes en eigenschappen van een State klasse. De klasse die het sleutel object vasthoudt, heeft toegang tot de State klasse van de betreffende StatefulWidget, via de *currentState* eigenschap van de GlobalKey.

Twee belangrijke voorbeelden van het gebruik van een GlobalKey zijn de combinaties met de *AnimatedListState* en de *FormState*.

De AnimatedListState is de State klasse van de reeds bestaande *AnimatedList* Widget. Dit is een Widget die toelaat dat zijn items op een meer visuele manier in de user interface terecht komen tijdens het toevoegen of verwijderen.

De FormState daarentegen, is de State klasse van de *Form* Widget. Deze Widget stelt een invulformulier voor. De FormState biedt de mogelijkheid om, bij Android gerelateerde Widgets, de validatie functies van alle onderliggende Material Design tekstvelden in één keer aan te roepen. De reden dat dit enkel werkt bij Android Widgets is te vinden in het verschil tussen de design guidelines. De Android specificatie voorziet voor tekstinvoer immers een tekstveld onder de invoervelden zelf, dit tekstveld dient voor waarschuwingsmeldingen en hints voor de gebruiker. De IOS specificatie voorziet voor tekstinvoer echter geen ingebouwd tekstveld voor de foutmeldingen.

De Form State zorgt naast deze extra functionaliteit ook voor het behouden van de invoer van het formulier bij configuratieveranderingen, wat mooi meegenomen is.

4.5 InheritedWidget

Naast de twee basis Widgets voor de user interface, bevat Flutter nog een andere belangrijke Widget. User interfaces volgen, net zoals bij native Android, ook het concept van een boomstructuur. Het gebruik van een boomstructuur kan in sommige gevallen wel leiden tot een probleem: *'Wat als een bepaalde Widget nood heeft aan een voorouder die*

zich ergens in de bovenliggende boom bevindt?'

Het gebruik van de *InheritedWidget* lost dit probleem op. De InheritedWidget biedt zijn (klein)kinderen de mogelijkheid om een voorouder te zoeken. Voor de InheritedWidget is er binnen Flutter een conventie: InheritedWidgets moeten een methode bevatten op klasseniveau die de eerstvolgende voorouder van het juiste type zal ophalen uit de boom. Deze methode volgt de signatuur *'static <type InheritedWidget> of(BuildContext context)'* en zal aan de context vragen om de voorouder op te halen aan de hand van de methode *dependOnInheritedWidgetOfExactType<type InheritedWidget>()*.

Codevoorbeeld 4.3: Het definiëren van een eigen InheritedWidget. Deze implementatie voorziet een stuk tekst aan zijn kinderen. Een extra aandachtspunt is hier de aanwezigheid van de eerder besproken Widget sleutels. (Bracke, 2020)

```
class MyWidget extends InheritedWidget {
  const MyWidget({
    Key key,
    @required this.data,
    @required Widget child,
  }): assert(data != null && child != null), super(key: key);

  final String data;

  static MyWidget of(BuildContext ctx)
  => ctx.dependOnInheritedWidgetOfExactType<MyWidget>();


  @override
  bool updateShouldNotify(MyWidget oldWidget)
    => data != oldWidget.data;
}
```

Het patroon van de InheritedWidget komt in Flutter vrij veel voor. Voorbeelden van implementaties zijn bijvoorbeeld de *Theme*, *Localizations* etc.

4.6 Gebruiksklare componenten - Android versus Flutter

De volgende secties zullen een aantal user interface elementen, die zowel native Android en Flutter aanbieden, vergelijken.

4.6.1 Toolbar versus AppBar

Native Android en Flutter bieden beide een component aan die wordt gebruikt voor de titel van een scherm en eventuele extra acties te tonen. Deze component wordt in native Android de *Toolbar* genoemd en Flutter biedt deze component aan onder de naam *AppBar*.

Een Toolbar opzetten in native Android is in vergelijking met Flutter wel wat meer werk. Enkel de layout uitschrijven is immers niet genoeg. De Toolbar moet ook ingesteld worden in de Activity, door een specifieke eigenschap in te stellen. Pas vanaf dit moment is de Toolbar klaar voor gebruik binnen de Fragments. Indien Fragments nood hebben aan de Toolbar, moeten deze altijd via een methode in hun Activity de toegang tot de Toolbar verkrijgen. Een extra nadeel aan de standaard Toolbar, is het feit dat deze enkel maar een titel aanbiedt. Het gebruik van een uitklapmenu is dan weer een pluspunt. Dit uitklapmenu kan namelijk gebruik maken van het situatiebewuste layout systeem van native Android. Dit heeft als resultaat dat zulke menu's zich makkelijk aanpassen aan het scherm van de gebruiker.

Het opzetten van een AppBar in Flutter, begint bij het vasthangen van deze widget aan een *Scaffold*. Op vlak van layout is de AppBar iets flexibeler dan de Toolbar. De AppBar biedt een aantal eigenschappen aan waar Widgets in kunnen worden gezet. Door deze eigenschappen te combineren kunnen layouts worden gemaakt die flexibeler zijn in vergelijking met wat een Toolbar out-of-the-box aanbiedt. Op vlak van uitklapmenu's is het dan net iets moeilijker bij de AppBar. Om het mogelijk te maken dat een uitklapmenu van een AppBar zich kan aanpassen op basis van de layout van het fysieke scherm (hier moet vooral worden gedacht aan de oriëntatie en schermgrootte), zijn developers namelijk genoodzaakt om zelf wat extra code te schrijven. In tegenstelling tot native Android is de AppBar dan weer direct te gebruiken, zonder enige configuratie.

Om het opzetten van een toolbar met een geïntegreerd opties menu te evalueren, zal er naar een implementatie van de volgende situatie worden gekeken.

- er is een titel
- er is een options menu
- er zijn 5 items in het menu en elk van deze items hebben als waarde een stuk tekst
- wanneer op een item wordt geklikt, verandert de titel van de toolbar naar het stuk tekst van het item
- de layout van de Toolbar/AppBar past zich aan voor een tablet of telefoon

Het uitschrijven van een xml-layout voor de Toolbar component, kon worden gerealiseerd met 14 lijnen code. Om het uitklapmenu van de Toolbar van opties te kunnen voorzien, werden de opties uitgeschreven in een extra xml-bestand. De code van dit extra bestand, voegt een extra 32 lijnen code toe aan het totaal. Hier werd ook de *app:ShowAsAction* eigenschap gebruikt. Dankzij deze eigenschap kan native Android het menu aanpassen aan de schermgrootte en oriëntatie. Om het selecteren van de opties in een Fragment te kunnen faciliteren, werd er nog een implementatie voorzien van *onOptionsItemSelected()*. Deze implementatie kost een extra 18 lijnen code. Om het Android systeem te laten weten dat er een Fragment is dat een uitklapmenu nodig heeft, werd er nog een configuratie geschreven van 4 lijnen code in dit Fragment. Ten slotte kost het instellen van de Toolbar in de Activity nog één extra lijn code. Indien al deze onderdelen worden opgeteld, kan er worden besloten dat de volledige implementatie een totaal van 69 lijnen code heeft gekost.

De implementatie van de AppBar voor de Flutter app bestond, net zoals in native Android,

uit een aantal onderdelen. Om de AppBar te kunnen gebruiken in een applicatie, moet deze worden klaargezet in de *Scaffold* widget, dit kost één lijn code. Om een equivalente implementatie te voorzien voor de *app:showAsAction* eigenschap van native Android, kan een StatefulWidget worden gebruikt. Deze zal instaan voor het aanpassen van de inhoud van de AppBar, op basis van de schermdimensies en oriëntatie. Het uitschrijven van deze StatefulWidget kost een totaal van 54 lijnen code. Ten slotte kan voor het selecteren van de keuzes nog een extra methode worden geschreven. Aangezien deze enkel de titel hoeft in te stellen, zoals vastgelegd in de test case, voegt deze methode maar één lijn extra toe aan het totaal. Als alle onderdelen van deze implementatie worden opgeteld, komt het totaal aantal lijnen code op 56 te staan. Hier valt vooral op te merken dat hoewel het aantal lijnen code wel minder is in de Flutter app, er wel een volledige StatefulWidget moet worden uitgeschreven. Deze StatefulWidget is wat boilerplate.[2]

4.6.2 RecyclerView versus GridView

Sommige applicaties maken gebruik van layouts die hun elementen in een raster onderverdelen. In deze sectie zal de implementatie van het keuzescherm, dat zelf een raster is, worden besproken.

Om raster-layouts te kunnen tonen op het scherm, maakt native Android gebruik van de *RecyclerView* component. Deze component kan op een performante manier veel elementen op het scherm tonen. Om de verschillende items in de lijst te kunnen tonen is het noodzakelijk dat er een layout wordt uitgeschreven die zal worden gebruikt om de elementen visueel voor te stellen in de lijst. Aangezien er verschillende layout mogelijkheden bestaan, moet de component nog worden geconfigureerd om een *raster*-layout te gebruiken.

Om een raster-layout te kunnen uitwerken in Flutter, kan de *GridView* Widget worden gebruikt. Deze Widget zal zijn items in een raster op het scherm tonen. Voor de layout van de verschillende items wordt best een specifieke Widget uitgeschreven, net zoals er bij native Android een aparte layout wordt voorzien.

Als voorbeeld van een raster-layout, kan er worden gekeken naar de implementatie van het raster zoals te zien in figuur 3.1.

Het implementeren van de xml-layout van de RecyclerView, kost 8 lijnen code. Om de items van een layout te voorzien werd er nog een extra layout bestand uitgewerkt. Deze layout file, eist 34 lijnen code. Het uitschrijven van de bijkomende data adapter, voegt nog eens 8 lijnen code toe aan het totaal. Deze adapter zal de data met deze component verbinden en is een noodzakelijke component. Ten slotte moet de RecyclerView nog worden geconfigureerd met deze adapter en moet er worden gekozen voor een raster layout. Deze configuratie kan worden uitgeschreven met 5 lijnen code. Als al deze componenten worden opgeteld, komt het totaal op 55 lijnen code te staan.

[2]De term 'boilerplate' slaat op een stuk extra code, dat nodig is om iets te kunnen implementeren, ten koste van de efficiëntie van een developer.

Om een GridView te implementeren in Flutter moet deze Widget worden gedefinieerd in de layout van een scherm. Ook moet deze Widget weten welke layout er zal worden gebruikt voor de verschillende elementen, dit kost in totaal 4 lijnen code. Het implementeren van een tekst-met-padding Widget dat zal dienen voor de item-layout kost een 12-tal lijnen code. Indien al deze componenten worden opgeteld, komt dit neer op een totaal van 16 lijnen code.

Hier valt vooral te concluderen dat het uitschrijven van de connectie tussen de effectieve data en de user interface, veel meer werk is in native Android in vergelijking met Flutter.

4.6.3 RecyclerView versus ListView

Om een groot aantal elementen voor te stellen in een applicatie kan ook een lijst worden gebruikt.

Lijsten in native Android verschillen niet zo veel van de raster layout. Om een lijst voor te stellen wordt ook hier gebruik gemaakt van de *RecyclerView* klasse. Het enige verschil tussen lijsten en rasters is het type van layout waarmee de RecyclerView wordt geconfigureerd. Op vlak van code zijn de lijsten en rasters equivalent (de effectieve layouts voor elementen kunnen wél verschillen, maar dit is hier irrelevant).

Ook de lijsten in Flutter hebben het voordeel van niet veel te verschillen met de raster layouts. Flutter voorziet voor lijst gebaseerde user interfaces de *ListView* Widget. Deze widget maakt gebruik van hetzelfde idee als de *GridView* om de layouts te voorzien voor de verschillende elementen.

Een ListView implementeren kost één lijn minder in vergelijking met een GridView, omdat er één eigenschap minder moet worden ingesteld, terwijl de rest kan worden overgenomen uit de implementatie van de GridView.

4.6.4 ViewPager2 versus PageView

Een laatste manier om een collectie van elementen te tonen aan een gebruiker is het gebruik van een 'Pager'. Het woord Pager is afgeleid van het Engelse 'Page', wat een pagina betekent. Dit soort user interface element wordt gebruikt om elementen te tonen in de vorm van een carrousel. Hierbij worden de elementen horizontaal of verticaal naast elkaar gezet en krijgen deze doorgaans meer plaats dan in een gewone lijst. De gebruiker kan er ook voor kiezen om naar links of naar rechts te scrollen, indien dit mogelijk is.

Native Android biedt voor dergelijke 'pagers' een component aan in de Android Jetpack library. De *ViewPager2* is een moderne implementatie die developers de mogelijkheid geeft om een pager te ontwikkelen. Achter de schermen maakt deze component gebruik van een aantal concepten die ook worden gebruikt bij de *RecyclerView*.

Het uitwerken van de xml-layout van een ViewPager2 omvat een totaal van 8 lijnen code. De adapter die de data zal verbinden met de user interface is van dezelfde soort zoals

bij rasters en lijsten, de resultaten van sectie 4.6.3 kunnen hier overgenomen worden. Hetzelfde geldt voor de layout van een enkel item. Het vasthangen van een ViewPager2 aan de achterliggende data, kost 6 lijnen code, wat het totaal van *niet herbruikbare code* op 14 lijnen brengt.

Flutter biedt een implementatie van een pager aan via de *PageView* Widget. Deze Widget volgt, op vlak van het aanmaken van elementen, hetzelfde idee als de *ListView* en *GridView*.

Om de huidige pagina (het element in het midden, zoals te zien in figuur 4.2) in de PageView te kunnen onderhouden is er wat configuratie nodig bij het definiëren van de PageView zelf. Deze configuratie kost 7 lijnen code. Naast deze configuratie moet de widget ook in de layout worden gezet, wat 3 extra lijnen code met zich meebrengt. Dit brengt het totaal aantal lijnen *niet herbruikbare code* op 10 lijnen. De code om items aan te maken is van dezelfde soort als bij de *ListView* en *GridView*, deze is herbruikbaar.

Ook hier komen de nadelen van de data adapters van native Android naar boven. Het uitschrijven van dergelijke adapters is relatief gezien veel extra werk en maakt het onderhouden van zulke componenten moeilijker.

Carrousel met de focus in het midden

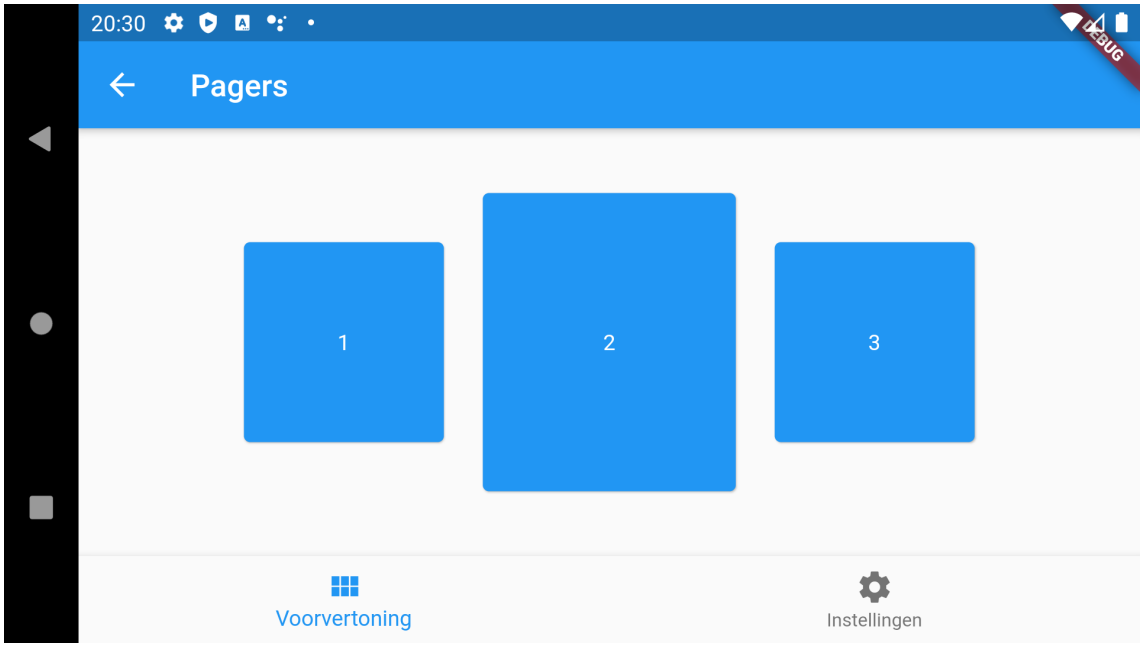
Als uitbreiding voor de ViewPager/PageView, kan er ook gekeken worden naar een speciale soort layout die populair is voor dit soort carrousel. Dit soort layout geeft het middelste element meer focus, door het groter te maken en de andere elementen te verkleinen. De elementen direct links en rechts van het middelste element worden nog voor een fractie van hun volledige grootte op het scherm getoond.

In native Android wordt voor dit soort layouts de respectievelijke ViewPager2 een stuk uitgebreid. Deze component geeft aan hoe een bepaalde pagina moet gemanipuleerd worden, wanneer deze op het scherm komt. Een concrete implementatie van een dergelijke uitbreiding kost 12 lijnen code.

Ook in Flutter is het mogelijk om een dergelijke layout te ontwerpen. Hiervoor wordt er gebruik gemaakt van een stuk functionaliteit dat wordt aangeboden door de PageView widget, samen met een *AnimatedBuilder*(voor een vlotte overgang tussen de pagina's). Een volledige implementatie van een carrousel, kost in Flutter een 17-tal lijnen code. Figuur 4.2 illustreert een mogelijke implementatie van een carrousel in Flutter.

4.7 Invoer van een gebruiker

In mobiele applicaties is er soms nood aan invoer van een gebruiker. Hiervoor wordt er aan gebruikers gevraagd om een formulier in te vullen. In deze sectie zal er worden gekeken naar een aantal componenten die kunnen worden gebruikt om een formulier te maken. Naast de componenten die dit onderzoek zal vergelijken, zijn er uiteraard nog een



Figuur 4.2: Een implementatie van een carrousel in de Flutter applicatie met behulp van een PageView widget. (Bracke, 2020)

hele reeks beschikbare elementen, maar deze vallen buiten de scope van dit onderzoek.

4.7.1 Tekstveld

Een tekstveld wordt gebruikt om een gebruiker de mogelijkheid te geven om tekst in te voeren.

In native Android wordt er voor tekstvelden gebruik gemaakt van de *TextInputLayout*. Dit is een invoerveld voor tekst, dat ook de mogelijkheid biedt om invoerfouten aan de gebruiker te tonen.

De xml-layout van een TextInputLayout, kost 14 lijnen code. Echter is het zo dat deze component niet out-of-the-box de validatiefouten zal tonen aan de gebruiker. Hiervoor moeten ontwikkelaars een stukje extra code schrijven, waarmee de connectie tussen de component en de effectieve validatie kan worden opgezet. Deze code kost een extra 3 lijnen code. Om tekstveranderingen op te vangen is ook een stuk boilerplate nodig, wat nog eens 8 lijnen code aan het totaal toevoegt. Het maken van een tekstveld kost een totaal van 25 lijnen code.

Flutter biedt voor tekstvelden de *TextFormField* widget aan. In tegenstelling tot de TextInputLayout heeft deze widget een ingebouwde eigenschap, de *validator*. Deze zal de invoer valideren én de foutmeldingen onderaan tonen. Het instellen van een validator kan met een enkele lijn code. Het configureren van het uitzicht van deze widget is noch een voordeel noch een nadeel. De widget kan volledig worden aangepast qua stijl, maar hiervoor moet een minimaal stukje boilerplate worden geschreven. Als voorbeeld kan een

label worden genomen. Het instellen van een label kan in native Android via een eigenschap, wat in één lijn code past. Aangezien de decoratie van de TextFormField apart zit, kost het instellen van een label twee lijnen code. Dit label wordt ingesteld via de decoratie van het invoerveld. Om de effectieve tekst van het invoerveld te onderhouden is een apart object nodig, wat nog een extra 2 3 lijnen code vergt (afhankelijk van het feit of de startwaarde van de tekst elders moet worden opgehaald). Al deze onderdelen samen, plus het declareren van de widget zelf, brengt het totaal aantal lijnen code op 6.

4.7.2 Dropdown Menu

Soms moeten gebruikers van een applicatie kunnen kiezen uit een klein aantal elementen. Indien developers hiervoor kiezen om een uitklapmenu te gebruiken, bieden native Android en Flutter respectievelijk de *Spinner* en de *DropdownButton* aan.

De layout van een Spinner kan worden geïmplementeerd met 8 lijnen code. Dit omvat de Spinner zelf en ook een melding zoals 'Kies een Optie', die wordt getoond indien de gebruiker nog niks heeft geselecteerd. Het doorgeven van de mogelijke opties aan de Spinner én het monitoren van de selectie kost redelijk wat werk. Het instellen van de te kiezen items kost 9 lijnen code en om de selectie te kunnen monitoren moeten er nog een extra 4 lijnen code worden geschreven. Indien de situatie waar er niks werd geselecteerd ook moet worden opgevangen, kost dit een extra 2 lijnen code. Een Spinner volledig implementeren kost een totaal van 21 23 lijnen code.

Om een uitklapmenu te kunnen gebruiken in Flutter apps, is er de *DropdownButton* Widget. Dit is een Material Design widget die kan worden gezien als het evenbeeld van de Spinner uit native Android. Deze widget maakt ook gebruik van dezelfde concepten die gelden voor widgets zoals de *ListView*, om de keuzes te kunnen opbouwen. Het monitoren van de geselecteerde keuze is bij de DropdownButton een stuk gemakkelijker in vergelijking met de Spinner. Zowel een geselecteerd item alsook een lege selectie worden samen afgehandeld. Hiervoor kan er met één lijn code een functie worden ingesteld, die deze veranderingen dan zal onderscheppen. Het definiëren van een DropdownButton in de layout van een widget, kost voor een volledige implementatie 4 lijnen code. Dit omvat de startwaarde, het opbouwen van de items en het monitoren van de selectie.

4.7.3 Schuifbalk

Indien een gebruiker een getal moet kunnen kiezen tussen twee waarden, kan er worden gebruik gemaakt van een verschuifbare balk. Native Android voorziet hiervoor een component onder de naam *SeekBar*. Flutter biedt voor deze component de *Slider* widget aan.

De SeekBar en de Slider lijken op het eerste zicht wel identiek maar er zijn toch een aantal minieme verschillen. Zo is het achterliggende datatype verschillend, de Slider kan nauwkeurigere waarden toelaten, aangezien deze kommagetallen verwacht. De SeekBar daarentegen, is beperkt tot gehele getallen. Ook kan er pas vanaf Android 8 gebruik worden gemaakt van de minimum eigenschap van de SeekBar, terwijl de Slider wel beide

aanbiedt. Het Flutter framework ondersteund Android 4.1, wat het mogelijk maakt om de Slider widget te gebruiken voor lagere versies van Android.

Het implementeren van de layout voor een SeekBar kan worden gerealiseerd met 9 lijnen code. Om de input van de SeekBar te kunnen gebruiken, moeten er nog een extra 5 lijnen code worden geschreven om de invoer op te kunnen vangen.

Een Slider definiëren in de layout van een Widget, kost een totaal van 5 lijnen code. Dit omvat het instellen van de grenswaarden én het opvangen van de input, alsook het instellen van de startwaarde.

4.7.4 Aan-uit Knop

Indien er van de gebruiker wordt verwacht dat deze een optie aan of afzet, zoals een instelling binnen een applicatie, wordt er soms gebruik gemaakt van een aan-uit knop. Native Android en Flutter bieden hiervoor elk een component aan onder de naam *Switch*.

De Switch van native Android, bevat naast de knop zelf ook een label, dat voor de knop wordt geplaatst. De Switch van Flutter bevat echter geen label.

De layout van een Switch in native Android, kan worden uitgeschreven met 11 lijnen code. Dit omvat ook het label zelf. Indien het label op basis van de huidige staat van de knop moet worden ingesteld, is er wel wat extra code nodig. Dit zou bijvoorbeeld met de DataBinding library kunnen worden opgelost. Om de input van de switch op te vangen moet er nog een extra methode aan de Switch worden vastgehangen. Het vasthangen van zo een methode, kost ook weer één lijn code. Dit brengt het totaal op 12 lijnen code voor de Switch van native Android.

Een Switch definiëren voor gebruik in een widget boom, kost 3 lijnen code. Dit omvat het definiëren van de Switch zelf, het instellen van de startwaarde én een lijn code om de veranderingen op te vangen.

4.7.5 Radio Button

Soms is het nodig dat gebruikers een keuze moeten kunnen maken tussen verschillende opties, waarbij deze elkaar uitsluiten. Bijvoorbeeld om te kiezen tussen het model van een wagen in een webshop.

Hiervoor bieden native Android en Flutter respectievelijk de *RadioButton* en de *Radio* componenten aan. De RadioButton van native Android heeft een label, terwijl de Radio widget er geen heeft.

De layout van een enkele RadioButton in een native Android app, kost 7 lijnen code.

Om in native Android een exclusieve keuze te kunnen maken waarbij de RadioButtons elkaar uitsluiten, moeten deze samen in een *RadioGroup* zitten. De layout van een Ra-

dioGroup eist, naast de code voor de individuele RadioButtons, ook nog eens 7 lijnen code.

Om de selectie van gegroepeerde RadioButtons op te kunnen vangen, kan er worden gebruik gemaakt van een extra methode, gecombineerd met DataBinding. Dit eist ook nog eens twee lijnen code op.

Het definiëren van een Radio widget kost, in tegenstelling tot de RadioButton van native Android, maar 3 lijnen code. Dit omvat de startwaarde én het instellen van een methode die de veranderingen kan opvangen.

De Radio widget van Flutter, pakt het idee rond een groep Radio widgets, iets slimmer aan. De gegroepeerde widgets, worden elk met een gedeelde waarde verbonden. Hierin zit dan de waarde van de geselecteerde knop. Indien er een aantal Radio widgets samen moeten worden gezet, kost dit per Radio 4 lijnen code plus één extra lijn, aangezien de gedeelde waarde moet worden ingesteld.

4.7.6 Selectievakje

De laatste component die zal besproken worden is het selectievakje. Deze component is vooral nuttig in het geval dat een gebruiker meerdere **niet uitsluitende** keuzes kan kiezen.

Native Android voorziet hiervoor de *CheckBox* en Flutter biedt dit soort component de *Checkbox* Widget aan.

Ook hier verschillen de twee met elkaar op vlak van het label dat wel aanwezig is bij native Android.

Aangezien deze component functioneel identiek is aan de *Switch* en enkel verschilt op vlak van uitzicht, is de implementatie in native Android en Flutter identiek hetzelfde aan deze laatste. Voor deze component geldt ook het resultaat uit sectie 4.7.4.

4.8 Animaties

In deze sectie zal worden gekeken hoe er gebruik kan gemaakt worden van animaties in native Android en in Flutter.

4.8.1 Animaties in native Android en Flutter

Om animaties te kunnen implementeren in native Android, kan er worden gekeken naar *Property Animations*. De naam Property Animations verwijst naar het animeren van de eigenschappen van een object. Om een eigenschap te kunnen animeren, kan er beroep worden gedaan op de *ObjectAnimator* klasse. Objecten van deze klasse zijn in staat om een enkele eigenschap te animeren van een startwaarde naar een finale waarde. Het ge-

bruik van deze klasse heeft als voordeel dat het animeren van een eigenschap vrij snel kan worden opgezet. Als voorbeeld kan een animatie van een enkele eigenschap worden genomen. Het maken van twee variabelen voor de start en eindwaarden, kost twee lijnen code. Het maken van de ObjectAnimator zelf kost een lijn code. Het instellen van de tijdsduur samen met het effectief starten van de animatie, kost nog eens twee lijnen code. Ten slotte moet de versie van het Android systeem nog worden nagekeken, bepaalde functionaliteit van de ObjectAnimator wordt maar ondersteund vanaf Android 5. Dit voegt nog eens een lijn code toe, wat het totaal op 6 lijnen code brengt voor een dergelijke animatie. Een extra voordeel aan de ObjectAnimator is het feit dat deze gemakkelijk kunnen worden samengevoegd, om meer complexe animaties te maken.

Het animatiesysteem van Flutter is iets uitgebreider dat dit van native Android. Er bestaan namelijk *impliciete* en *expliciete* animaties. Ook bestaan er een aantal libraries die het bestaande systeem nog kunnen aanvullen, zoals Flare. In dit onderzoek zal enkel worden gekeken naar de expliciete animaties. Deze kunnen worden beschouwd als het evenbeeld van de Property Animations uit native Android.

Het implementeren van een expliciete animatie in Flutter is meer werk, in vergelijking met native Android. Om een *expliciete* animatie van een enkele eigenschap te kunnen implementeren, kan enkel een StatefulWidget worden gebruikt, aangezien er expliciet zal worden aangegeven wanneer de animatie moet starten. Het maken van deze widget kost 3 lijnen code. Naast het aanmaken van de widget, moeten er ook een aantal variabelen worden aangemaakt, dit kost dan weer 4 lijnen code. Om de animatie in de widget boom te kunnen gebruiken moet een *AnimatedBuilder* worden gedeclareerd. Het opzetten van deze widget kost 4 lijnen code. Dit omvat 3 lijnen voor de AnimatedBuilder zelf en 1 lijn voor de functie die, op basis van de huidige waarde van de animatie, de geanimeerde widget zal opbouwen. Ten slotte moet de animatie nog worden gestart, wat ook een lijn code vraagt. Als alle componenten worden samengeteld, kost een expliciete animatie van een enkele eigenschap 12 lijnen code.

Een nadeel van het animatiesysteem van Flutter is het feit dat het gebruiken van meerdere animatie objecten naast elkaar (om bijvoorbeeld twee eigenschappen te animeren) wel meer code kost, in vergelijking met native Android. Wat native Android echter niet heeft is een gemakkelijke manier om een *curve* te definiëren voor de animatie. Flutter biedt hiervoor een speciale animatieklasse aan, waar een dergelijke curve kan worden opgegeven als argument. Een bijkomend voordeel is het feit dat de AnimatedBuilder in bepaalde gevallen de animatie kan optimaliseren. Indien een stuk van de te animeren widget boom niet veranderd tijdens de animatie, kan dit stuk eenmaal worden doorgegeven, om later te hergebruiken. Hier kan bijvoorbeeld worden gedacht aan het vergroten van een widget, de inhoud van de widget zal niet veranderen maar zijn aandeel binnen de layout wél. Wat ook in het voordeel spreekt van Flutter is het feit dat *alle* te animeren eigenschappen worden ondersteund op dezelfde versies van Android, in tegenstelling tot native Android. Een belangrijk voorbeeld hiervan is het animeren van kleuren. De methode die een RGBA[3] ObjectAnimator maakt, is niet beschikbaar voor toestellen die niet op Android 5 of hoger

[3]Kleuren worden voor computers voorgesteld als een combinatie van vier getallen. Deze stellen respectievelijk de fractie rood, groen, blauw en de mate van transparantie voor. Daarom dat de term Red, Green, Blue and Alpha of kortweg RGBA, in het leven werd geroepen. (Eck, 2016)

draaien. De overeenkomstige *ColorTween* klasse in Flutter is wél toegankelijk op lagere versies van Android.

4.8.2 Animeren van gedeelde elementen

In bepaalde gevallen is het soms interessant om een visueel element uit het huidige scherm mee te nemen naar het volgende scherm. Hierbij wordt een bepaald element gekozen als het 'anker' voor de overgang.

In native Android wordt dit soort overgangen *Shared Element Transitions* genoemd. Om een Shared Element Transition te implementeren moet er wel wat configuratie gebeuren. De elementen die zullen worden gebruikt voor de overgang, moeten worden gemarkeerd via een speciale eigenschap. Dit kost *per variant van een layout* één lijn code. Voor een minimaal voorbeeld met een start en eindbestemming, waarbij er geen extra varianten zijn van de layouts, kost dit twee lijnen code. Dit heeft natuurlijk als nadeel dat dit zorgt voor meer dubbele code, naarmate er meer layout varianten worden uitgeschreven. Bij het oproepen van de effectieve navigatie, moet er een extra object worden meegegeven. Het maken van dit object kost een extra lijn code. De navigatie zelf opstarten kost ook één lijn code. Ten slotte moet in de *bestemming* nog een eigenschap worden ingesteld, anders zal er geen animatie plaatsvinden. Deze eigenschap instellen kost nog een laatste lijn code, wat het totaal op 5 lijnen code brengt.

Om een overgang met een gedeeld element te kunnen faciliteren binnen het Flutter framework, bestaat er een specifieke widget. De *Hero* Widget markeert zijn kind als geschikt om een overgang uit te voeren waarbij dit kind als anker wordt gebruikt. Het configureren van de twee Hero widgets voor het vertrekpunt en de eindbestemming, kost 3 lijnen code per Hero widget. Ten slotte moet de navigatie naar de eindbestemming worden opgestart. Dit kost één lijn code, wat het eindtotaal op 7 lijnen brengt. Een nadeel van de Hero widget is het feit dat, bij Android apps, het kind van de Hero widget zal moeten worden genest binnen een *Material* widget. De stijl informatie gaat immers verloren binnen de transitie.

4.9 Oriëntatie

De oriëntatie van een toestel is de positie waarin de gebruiker zijn toestel vasthoudt. De gebruiker kan zijn toestel vasthouden op een manier waar de langste zijde verticaal wordt gehouden. Dit heet de Portrait oriëntatie. Naast de Portrait oriëntatie kan een gebruiker een toestel ook vasthouden op een manier waar de langste zijde horizontaal ligt. In dit geval bevindt het toestel zich in de Landscape oriëntatie.

De oriëntatie van een toestel is iets waar layouts (soms) rekening mee moeten kunnen houden. Daarom leveren zowel native Android alsook Flutter een manier aan om met de oriëntatie van een toestel om te gaan.

4.9.1 Oriëntatie in native Android en Flutter

Op vlak van oriëntaties doet het systeem van native Android veel werk voor de developers. Developers moeten enkel de beperking op de oriëntatie instellen voor de respectievelijke layout bestanden. Het systeem zal zelf de juiste bestanden gebruiken op het gepaste moment. Indien er bijvoorbeeld een variant voor de landscape oriëntatie moet worden gebruikt, wordt de beperking voor deze oriëntatie op de respectievelijke layout bestanden gezet. Dit systeem kan veel meer dan een beperking op de oriëntatie afhandelen en het werkt ook met andere soorten van applicatie bestanden, zoals bijvoorbeeld vertalingen. Developers moeten 'weinig' extra doen, behalve de layouts zelf opstellen(wat relatief gezien wél veel werk kan zijn) en de beperkingen definiëren op het moment dat er een nieuw layout bestand wordt aangemaakt.

In Flutter wordt de layout niet door het systeem zelf geregeld. Developers zijn genoodzaakt om zelf de oriëntatie te bepalen via de informatie uit de *MediaQuery* Widget. Dit is een InheritedWidget die alle informatie bevat over het toestel van de gebruiker. Er bestaat ook een speciale Widget, de *OrientationBuilder*. Deze bevat ook informatie over oriëntatie. Er is echter een belangrijk verschil tussen de oriëntaties van MediaQuery en OrientationBuilder. De oriëntatie van MediaQuery is die van het toestel, terwijl de OrientationBuilder kijkt naar de *lokale* afmetingen en niet de globale van het toestel.

Om een layout te maken die zich aan zal passen aan de oriëntatie zijn er 4 lijnen code nodig per Widget. Dit omvat het nagaan van de oriëntatie via de MediaQuery en het aanroepen van een methode die een aangepaste Widget boom terug geeft.

Indien er met de ingebouwde OrientationBuilder moet worden gewerkt, omwille van de *lokale* limieten, kan de layout worden gemaakt met 6 lijnen code. Dit omwille van het feit dat de OrientationBuilder zelf ook moet worden gedefinieerd in de layout.

Codevoorbeeld 4.4: Het bepalen van de oriëntatie, aan de hand van de MediaQuery Widget. De methodes voor portrait en landscape kunnen een andere layout voorzien. (Bracke, 2020)

```
class MyWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    if(MediaQuery.of(context).orientation
                == Orientation.portrait){
      return portrait();
    }else{
      return landscape();
    }
  }
}
```

4.10 Schermdimensies

Soms is het nodig dat bepaalde layouts in een applicatie zich kunnen aanpassen aan de dimensies van het scherm. Native Android en Flutter hebben beide oplossingen om hiermee rekening te houden.

Native Android gebruikt voor het aanpassen van de layouts naar andere schermdimensies, dezelfde manier van werken zoals bij de oriëntatie. Ook hier worden de layout bestanden opgedeeld, ditmaal via een beperking op de minimum breedte van een scherm, en zal het systeem bepalen welke layout er moet worden gebruikt.

Wat Flutter betreft moeten de developers ook hier de informatie zelf ophalen uit de MediaQuery Widget. Deze bevat de schermdimensies van het toestel. Dit is ongeveer hetzelfde als het gebruik van de breakpoints uit web development. De term 'breakpoint' slaat op de breedte van een schermresolutie die de scheiding tussen twee verschillende layouts voorstelt. Als voorbeeld zou een resolutie met een breedte van 768 pixels kunnen worden genomen. Dit is het punt waarop het scherm (meestal) overgaat van een telefoon resolutie naar een tablet-achtige resolutie. (Otto, 2015)

Codevoorbeeld 4.5: Layouts kunnen zich aanpassen aan de schermbreedte, aangegeven door de huidige *MediaQuery*. (Bracke, 2020)

```
class MyWidget extends StatelessWidget {

    @override
    Widget build(BuildContext context){
        final width = MediaQuery.of(context).size.width;
        final someBreakpoint = 768;

        if(width < 768){
            return someLayout();
        }
        ....
    }
}
```

4.11 Voor- en Nadelen van beide systemen

Nu beide systemen geëvalueerd werden op vlak van user interface, is het misschien handig om de meest uitgesproken verschillen rond user interface ontwikkeling eens op een rij te zetten. Zoals te zien in tabel 4.1 is Flutter superieur in het maken van de meeste user interface elementen. Langs de andere kant is Android beter in het scheiden van verschillende schermconfiguraties. Op vlak van animaties biedt Android een snellere ontwikkeling aan.

Tabel 4.1: Voordelen van user interface ontwikkeling in Flutter en Android (Bracke, 2020)

Flutter
Lijst-achtige User Interfaces zijn eenvoudiger
Betere verbinding tussen UI en business logica
User interface componenten zijn meer modulair
Meer mogelijkheden voor animaties
Shared Element Transitions zijn heel eenvoudig
Native Android
Betere uitvouwmenu's in een Toolbar
Layouts maken voor verschillende schermen is eenvoudig
Layouts maken voor verschillende oriëntaties is eenvoudig
Animaties opzetten gaat sneller

Tabel 4.2: Een aantal Cupertino equivalenten van Material Widgets (Bracke, 2020)

Material	**Cupertino**
MaterialApp	CupertinoApp
Scaffold	CupertinoPageScaffold
AppBar	CupertinoNavigationBar
BottomNavigationBar	CupertinoNavigationBar
RaisedButton	CupertinoButton

4.12 Cupertino Widgets

Hoewel dit onderzoek voor de onderzoekscriteria zich toespitste op de Material Widgets, wil dit niet zeggen dat er geen Widgets bestaan die een andere stijlgids volgen.

Naast de Widgets uit de Material Design stijlgids bestaat er ook een verzameling Widgets voor het IOS platform. Deze Widgets worden de Cupertino[4] Widgets genoemd. In tabel 4.2 is een kleine selectie van deze Widgets te vinden, naast hun equivalent uit de Material Design stijlgids.

De *CupertinoApp* Widget is een widget waarvan kan gestart worden indien een app met IOS stijlen nodig is. Deze biedt, net zoals de *MaterialApp* een *Navigator* Widget aan, samen met een *CupertinoTheme*, een thema widget voor IOS. De *CupertinoPageScaffold* is het IOS equivalent van een *Scaffold*, deze widgets stellen een 'pagina' voor. De CupertinoPageScaffold heeft plaats voor de IOS equivalenten van de *AppBar* en *BottomNavigationBar*. De *RaisedButton* van een Android app kan worden vervangen door de *CupertinoButton* indien knoppen nodig zijn, volgens de IOS stijlgids.

Het bepalen van het huidige platform gebeurt via de *Platform* klasse. Zo kan een platform specifieke widget boom de user interface voorstellen. Dit is echter geen verplichting. De

[4]Cupertino verwijst naar de plaats waar Apple Inc. is gevestigd.

Skia renderer[5] is zelf niet afhankelijk van het platform en kan perfect een MaterialApp widget op een Iphone laten zien.

[5]Een renderer is een object dat de user interface op het fysieke scherm zal tekenen. De term *renderer* is afgeleid van het Engelse werkwoord *to render*, wat tekenen betekend.

5. Asynchroon Werk

5.1 Inleiding

Bij veel applicaties is het soms nodig dat er een bewerking wordt opgestart die wel wat tijd in beslag kan nemen. Denk aan het ophalen van een afbeelding van een website of het laden van berichten op sociale media. Deze bewerking door de *Main Thread* van een applicatie laten uitvoeren zal leiden tot de zogenaamde *App Not Responding* fouten. De Main Thread is het hoofdproces van een applicatie en indien deze door een langdurige bewerking te veel werk krijgt zal de applicatie trager verdergaan. Om toch grote bewerkingen uit te kunnen voeren kunnen applicaties beroep doen op *asynchrone* bewerkingen. Deze maken gebruik van een apart proces om de langdurige bewerking afzonderlijk van de Main Thread uit te voeren. Veel talen zoals Javascript en C# hebben hiervoor ingebouwde features. (Goranova, Kalcheva-Yovkova & Penkov, 2015)

In dit hoofdstuk zal er worden gekeken naar de manieren die native Android en Flutter aanbieden om met zulke asynchrone bewerkingen te werken.

5.2 Coroutines versus Futures

Asynchrone bewerkingen kunnen worden opgedeeld in twee categorieën. Enerzijds bestaan er de *single-value* bewerkingen, deze geven een *enkele* waarde terug na het uitvoeren van hun bewerking. Anderzijds zijn er ook de *multiple-value* bewerkingen, deze zijn in staat om meerdere waarden terug te geven over een bepaalde periode.

Deze sectie zal verder ingaan op de enkelwaardige bewerkingen en in sectie 5.3 zullen de

bewerkingen met meerdere waarden worden besproken.

Native Android apps maken voor enkelvoudige asynchrone bewerkingen gebruik van *Coroutines*. Dit is het Kotlin equivalent van een dergelijke asynchrone bewerking. (Luca Crisan, 2019)

Codevoorbeeld 5.1: Een Channel kan meerder waarden uitsturen. (Bracke, 2020)

```
suspend fun startChannel(channel: Channel<Int>){
    for (x in 1..5){
        delay(200L)
        channel.send(x * x)
    }
    channel.close()
}

suspend fun receiveFromChannel(channel: Channel<Int>){
    for (i in channel){
        value = i.toString()
    }
}
```

Een voordeel van deze Coroutines is het feit dat deze via een extensie op de *ViewModel* klasse vrij eenvoudig kunnen worden onderhouden, met oog op de huidige staat van de levenscyclus van de applicatie. Een nadeel van Coroutines is dan weer het feit dat eventuele fouten die kunnen optreden, manueel moeten worden opgevangen.

Dart, de programmeertaal voor Flutter apps, voorziet voor enkelvoudige asynchrone bewerkingen de *Future* klasse samen met de *async* en *await* keywords. (Belchin & Juberias, 2015) Conceptueel volgt Dart hetzelfde idee zoals Javascript en C#. Het afhandelen van de fouten van een Future gebeurt op een andere manier dan bij de Coroutines van native Android. Futures bieden immers de mogelijkheid om via een methode rechtstreeks de fout af te handelen.

Codevoorbeeld 5.2: Een asynchrone bewerking kan worden opgestart via een Future. Dankzij de eenvoudige syntax kunnen de fouten ook opgevangen worden. (Bracke, 2020)

```
Future<String> doAsyncOperation() async {
    return Future.value("someValue");
}

void awaitSomeFutureWithThen() async {
    return await doAsyncOperation().then((value){
        //use value
    }, onError: (err){
        //do something with err
    });
}
```

Het maken van een minimale Coroutine, met behulp van een aparte functie, kost 2 lijnen code. Het oproepen van deze Coroutine kost een extra 2 lijnen code. Dit omvat een lijn voor het opstarten van de bewerking en een lijn voor het wachten op het resultaat. Indien er voor de betreffende Coroutine een foutafhandeling moet worden geschreven, kan een foutafhandelingsblok, ook wel een *try/catch* blok genoemd, worden gebruikt. Een minimaal try/catch blok kost 3 lijnen code. Een volledige implementatie, met een try/catch blok en een coroutine functie, kost 7 lijnen code.

Het uitschrijven van een minimale Future via een aparte methode kost twee lijnen code. Het oproepen van deze Future en het resultaat binnenhalen, kan met één lijn code. Indien er voor deze Future een foutafhandeling moet worden geschreven kan dit worden gerealiseerd met een tweetal lijnen code. Een volledige Future uitschrijven kost in het totaal 5 lijnen code.

5.3 Channels versus Streams

Om een asynchrone bewerking te implementeren die meerdere waarden kan teruggeven over een bepaalde periode, voorziet native Android de *Channel* klasse. Channels zijn echter nog voor een deel een experimentele feature. Om een Channel te kunnen gebruiken moeten er twee Coroutines worden geschreven. De eerste Coroutine zal de Channel opvullen met waarden en de tweede Coroutine zal deze waarden uitlezen. Deze aanpak resulteert echter in een stuk boilerplate code, wat in het nadeel speelt van de huidige implementatie van deze feature.

Een minimale opstelling van een Channel kan worden gerealiseerd met 11 lijnen code. Het opzetten van een Channel object kost hiervan 5 lijnen code. Het ontvangen van de waarden van dit object kost 3 lijnen code. De huidige (nog steeds experimentele API) rond Channels bevat wel een kritieke fout. Indien de Channel wordt gesloten via een foutmelding, crasht de applicatie. Indien er een foutafhandelaar wordt geregistreerd op het Channel object zelf, vangt deze geen fouten op. Om dit op te lossen kan er worden besloten om de fouten **niet** door te geven via de *close()* methode maar een try/catch blok te gebruiken. Dit extra try catch blok, kost een extra 3 lijnen code.

Flutter biedt voor de asynchrone bewerkingen die meerdere waarden kunnen teruggeven, de *Stream* klasse aan. Deze klasse verschilt wat met de implementatie van de Channels uit native Android. Ten eerste bieden Streams een rechtstreekse optie aan om een element *of* een fout door te sturen. Een tweede punt is de eenvoudigere manier om elementen toe te voegen. Dit is gemakkelijk te realiseren door een overeenkomstige functie van de Stream aan te spreken, zijnde voor een nieuw element of voor een fout. Ten slotte stoppen Streams **niet** wanneer ze een fout ontvangen, in tegenstelling tot de Channels van native Android.

Codevoorbeeld 5.3: Een Stream kan worden gemanipuleerd via zijn *StreamController*. (Bracke, 2020)

```
//De RxDart BehaviourSubject is een extensie
//van de RxDart package
//voor de bestaande StreamController klasse.
final _sc = BehaviorSubject<String>();

void emitStreamValue(String value){
  _sc.add(value);
}

void emitStreamError(Object error){
  _sc.addError(error);
}
```

Een (minimale) Stream aanmaken kan worden gerealiseerd met één lijn code. Het toevoegen van een nieuw element kost één lijn code, terwijl het toevoegen van een fout ook één lijn code kost. Het luisteren naar de veranderingen van de Stream kost 4 lijnen code. Hierbij werd rekening gehouden met het opvangen van nieuwe data én foutmeldingen. Dit brengt het totaal van een complete Stream implementatie op 6 lijnen code. Deze resultaten werden bekomen via het gebruik van de *StreamController* klasse en niet via een zelfgeschreven *generator*[1] functie.

5.4 Asynchrone Widgets

Flutter biedt naast de *Future* en *Stream* klassen, ook specifieke widgets om deze twee te kunnen integreren in een user interface. De *FutureBuilder* en *StreamBuilder* widgets kunnen worden gebruikt om een stuk user interface te koppelen aan een Future of een Stream. Deze widgets maken het onderhouden van de achterliggende status van de beweringen heel gemakkelijk. De status van de huidige bewerking zit namelijk in een aparte klasse. Hiermee kan er op een eenvoudige manier worden bepaald welk soort user interface er op een gegeven moment tijdens de asynchrone bewerking moet getoond worden. Er kan gemakkelijk worden geschakeld tussen een widget met de ingeladen data, een foutmelding of een widget die aangeeft dat de bewerking nog bezig is. Dit is een verbetering ten opzichte van de werkwijze van native Android, waar er handmatig naar de status van de bewerking moest worden gekeken, om vervolgens bepaalde elementen te tonen of te verbergen.

[1] Generator functies worden gebruikt om handmatig Streams en *Iterables* te maken. Iterables zijn de niet asynchrone versies van Streams.

Codevoorbeeld 5.4: Een FutureBuilder definiëren die een Future object in de user interface integreert. (Bracke, 2020)

```
FutureBuilder(
    initialData: "Future not started",
    future: Future.value("Some async value"),
    builder: (context,snapshot){
      if(snapshot.connectionState == ConnectionState.done){
        if(snapshot.hasError){
          return MyErrorWidget(snapshot.error);
        }else{
          return MyDataWidget(snapshot.data);
        }
      }else{
        return MyLoadingWidget();
      }
    });
```

5.5 Conclusies uit dit hoofdstuk

Op vlak van asynchroon werk is er een significant verschil merkbaar tussen de twee systemen. Hier heeft het Flutter framework de bovenhand. Het biedt een stabielere manier om met dit soort features te werken. Ook wordt het integreren van user interfaces een heel stuk eenvoudiger.

6. Persistentie

6.1 Inleiding

Mobiele applicaties laten gebruikers meestal toe om bepaalde informatie lokaal op te slaan in een app. Hiervoor bieden applicaties een vorm van persistente opslag aan in de applicatie zelf. Dit hoofdstuk zal zich verdiepen in een aantal verschillende mogelijkheden die native Android en Flutter ter beschikking stellen om een dergelijke vorm van gegevensopslag te kunnen ontwikkelen.

6.2 Room versus Moor

Een eerste mogelijkheid voor een persistente oplossing binnen een applicatie zijn de *Room* en *Moor* databases.

Room is een lokale database voor native Android apps, die mooi aansluit bij andere componenten uit het Android ecosysteem. Zo heeft Room ingebouwde ondersteuning voor de eerder besproken Coroutines. Ook kan Room via *LiveData* objecten real-time dataveranderingen aanbieden. Het opbouwen van database queries is hetzelfde als het uitschrijven van traditionele SQL statements, wat het gemakkelijker maakt om hiermee aan de slag te kunnen. Room biedt ook de mogelijkheid om op een *type safe* manier parameters in queries te vermelden. Dit houdt in dat een ontwikkelaar geen waarden van een verkeerd type aan een query parameter kan geven. Aangezien Room een *relationele* database is, is er ondersteuning voor relaties tussen objecten. Om een complexe relatie op te zetten is er wel wat boilerplate nodig, wat niet echt optimaal is. Ook ondersteunt Room database transacties, wat het gemakkelijk maakt om verschillende queries op een veilige manier

samen uit te voeren. Room heeft echter geen ingebouwde ondersteuning voor datums. Om toch datums te kunnen opslaan in de database, zijn developers genoodzaakt om een conversie te doen naar tekst.

Codevoorbeeld 6.1: De DAO zorgt voor de connectie met de database tabellen. (Bracke, 2020)

```
@Dao
interface TestEntityDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertEntity(entity: TestDatabaseEntity)

    @Query("SELECT * FROM TestDatabaseEntity")
    suspend fun getEntities(): List<TestDatabaseEntity>

    @Delete
    suspend fun deleteEntity(entity: TestDatabaseEntity)

    @Update
    suspend fun updateEntity(entity: TestDatabaseEntity)
}
```

Room heeft ook een implementatie voor Flutter apps. Moor erft een aantal concepten over van Room. Moor is immers ook een relationele database. Net zoals Room ondersteunt Moor ook asynchrone bewerkingen. Aangezien Moor gebruik maakt van Streams en Futures, kan de achterliggende data worden geobserveerd via een Stream. Zo kunnen veranderingen in real-time worden opgevolgd. Moor verschilt wel op het vlak van DAO's[1]. Waar deze bij Room verplicht zijn, kunnen developers bij Moor kiezen of deze worden uitgeschreven of niet. In vergelijking met Room moeten ontwikkelaars zelf geen SQL schrijven, Moor zal deze achter de schermen genereren. De implementaties voor DAO's kunnen volledig in Dart worden geschreven. Waar Moor ook verschilt van Room is het gebruik van datums. Moor zal deze *zelf* omzetten. Dit is een verbetering ten opzichte van het handmatige werk in Room. De syntax om een relatie tussen twee tabellen te leggen is ook iets korter. Het definiëren van relaties blijft in beide systemen wel nog gevoelig voor fouten van developers. Beide implementaties eisen namelijk dat ontwikkelaars de gelinkte eigenschap **via een stuk tekst** instellen in de tabeldefinitie. Hier kan heel gemakkelijk een tikfout worden gemaakt, wat een ongelukkig nadeel is van beide systemen. Een laatste nadeel aan Moor is het feit dat de code generatie die de databaseklasse zal maken, handmatig moet worden opgestart via het *build_runner* commando. Room doet de generatie vanzelf, op het moment van het maken van een applicatie executable. Deze fase wordt ook wel de build fase genoemd.

Omdat Moor is afgeleid van Room, is het mogelijk om deze systemen met elkaar te vergelijken op vlak van implementatie. Concreet kan er worden gekeken naar het opzetten van de database klassen, het definiëren van een tabel én het opzetten van een DAO.

[1] DAO's zijn interfaces die de achterliggende database beschikbaar maken via een API.

Om de twee databases met elkaar te vergelijken, zal er worden gekeken naar een minimale situatie met één tabel.

Een implementatie van een Room database met één tabel, kost in totaal 16 lijnen code. Het uitschrijven van de databaseklasse kost 3 lijnen code. Om de database te kunnen manipuleren werd een DAO geschreven voor de enige tabel. Deze DAO ondersteunt de *CREATE*, *READ*, *UPDATE* en *DELETE* database commando's. Het opzetten van deze DAO eist 10 lijnen code op. Ten slotte bevat de database één tabel waarin de data wordt opgeslagen. Deze tabel omvat (in deze testcase) één eigenschap, de verplichte *primaire sleutel*. Het opzetten van een dergelijke tabel kost 3 lijnen code, plus één lijn per extra eigenschap in de tabel.

Het uitschrijven van een volledige implementatie met Moor kost 17 lijnen code. Hiervoor werd de Room implementatie nagebootst. Het definiëren van de database zelf kost 6 lijnen code. Het definiëren van een DAO kost in Moor 8 lijnen code. Deze ondersteunt ook het volledige spectrum CRUD bewerkingen. De DAO kan wel niet in een stap worden gegenereerd, wat een extra nadeel is van de huidige implementatie van Moor. De tabel wordt gedefinieerd met 2 lijnen code. Een extra eigenschap in de tabel toevoegen, kost één lijn code. Ten slotte wordt de link gelegd met de gegenereerde database, dit kost nog één extra lijn code.

6.3 Realm

Voor native Android apps bestaat er naast Room ook nog een ander alternatief, Realm. Realm is net zoals Room een relationele database. In vergelijking met Room, worden de databasequeries anders opgezet. Er worden namelijk geen expliciete database queries meer uitgeschreven met de SQL syntax. In de plaats van deze queries komt een meer object georiënteerde aanpak. Deze aanpak heeft als voordeel dat het doorgeven van parameters vrij eenvoudig kan gebeuren. Een bijkomend voordeel van deze aanpak is het feit dat de queries een stuk korter zijn, in vergelijking met een query van Room. Er zijn immers geen annotaties meer nodig. Ook maakt Realm geen gebruik van DAO klassen, wat het iets gemakkelijker maakt om met de database te werken, aangezien er een laag minder moet worden uitgeschreven. Realm heeft echter geen ingebouwde ondersteuning voor Coroutines, wat kan worden beschouwd als een minpunt voor deze library. Ten opzichte van Room verbetert Realm wel de manier om relaties in te bouwen in de database. Ook biedt Realm de mogelijkheid om een datum op te slaan, wat een verbetering is ten opzichte van Room. Een laatste punt dat zeker mag worden vermeld is het feit dat Realm geen *autoincrement primary keys* toelaat. In relationele databases is dit een identificatie-eigenschap van een element in een database tabel, waarbij elk nieuw element een sleutel krijgt die één waarde hoger ligt dan het vorige element. Het ontbreken van deze feature klinkt op het eerste zicht wel wat vreemd, aangezien dit een veelvoorkomende eigenschap is van elementen in andere relationele databases. Realm ondervindt hier echter geen hinder van.

6.4 Hive

Flutter biedt naast relationele databases ook een aantal opties aan om met *NoSQL* gebaseerde databanken te werken. *Hive* is een *key-value* datastore. Hierbij wordt er voor het model van de persistentie gebruik gemaakt van sleutels. Achter elke sleutel wordt een waarde opgeslagen. Tabellen in Hive worden *Boxes* genoemd[2]. Elke Box zal zijn waarden opslaan als een paar van een sleutel en een waarde. Naast de gewone Box klasse biedt Hive ook een alternatief aan, de *EncryptedBox*. Deze Box maakt het mogelijk om heel gemakkelijk de opgeslagen data te encrypteren. Een voordeel van Hive is het feit dat de achterliggende database weinig plaats in beslag neemt. Hive is echter niet zo geschikt om relaties in te bouwen in de data. Hoewel het effectief mogelijk is, via *HiveList*s, wordt het wel afgeraden voor complexere modellen. Op het eerste zicht lijkt Hive een overbodige oplossing, dit is echter een verkeerde assumptie. Hive is perfect geschikt om kleine, heel specifieke dingen, op te slaan. Hive zou bijvoorbeeld kunnen gebruikt worden als een cross-platform implementatie voor de *SharedPreferences* van Android.

6.5 Sembast

Flutter biedt naast key-value alternatieven zoals Hive ook andere soorten van NoSQL databanken aan. *Sembast* is een NoSQL-datastore die gebruik maakt van de Javascript Object Notation, kortweg JSON, om data op te slaan in tabellen. Sembast slaat, net zoals de Hive datastore, zijn persistente data op in tabellen. Sembast biedt echter een aantal extra features aan die ook te vinden zijn bij relationele databanken. Zo is er binnen deze library ondersteuning om database queries uit te voeren. Sembast hanteert hiervoor een object gebaseerde aanpak. De effectieve queries maken gebruik van de *Finder* klasse om te zoeken naar de data. Sembast ondersteunt ook het schrijven van queries die de data filteren via een bepaald criterium. Hiervoor kan de *Filter* klasse in de query worden opgegeven.

Codevoorbeeld 6.2: Een Sembast query met een Finder en een Filter. (Bracke, 2020)

```
final store = stringMapStoreFactory.store("store");

Future<SomeObject> findFirst(String someValue) async {
    final filter fl = Filter.equals("someField", someValue);
    final f = Finder(filter: fl);
    return await store.findFirst(_db, finder: f);
}
```

Het uitvoeren van database queries kan ook via transacties gebeuren indien gewenst. Het uitschrijven van Sembast queries kost wel wat code, aangezien Finder en Filter objecten aangemaakt moeten worden. Sembast biedt dankzij de integratie met Futures en Streams ook ondersteuning aan om in real-time te luisteren naar veranderingen in de data. Sembast ondersteunt ook het gebruik van pagination, via extra opties van de Finder klasse. Een

[2]Hive werd geïnspireerd door bijenkorven uit de natuur, vandaar de benaming.

Tabel 6.1: De verschillende voor- en nadelen van de beschikbare lokale databases. (Bracke, 2020)

Database	**Voordeel**	**Nadeel**
Room	Typesafe SQL queries	Duur qua opzet
Realm	Tabel relaties zijn eenvoudig	Geen ondersteuning voor Coroutines of Live Data
Moor	Gegenereerde DAO's	Codegeneratie manueel starten
Sembast	Zeer rijk aan features	Duur qua opzet
Hive	Snel op te zetten	Zeer beperkte use case

laatste punt dat zeker vermeld mag worden is het feit dat Sembast, omwille van de opslag met JSON, data relaties niet zelf zal modeleren. Ontwikkelaars zijn zelf genoodzaakt om een tussentabel te maken en deze dan te gebruiken om een relatie te modelleren in de database.

7. Navigatie

7.1 Inleiding

Een mobiele applicatie bestaat uit meerdere schermen die elk een deel van de totale functionaliteit aanbieden. Een gebruiker moet in staat zijn om tussen de verschillende schermen of *bestemmingen* te kunnen schakelen. Dit schakelen tussen schermen vormt de *navigatie* binnen een app. Dit hoofdstuk zal de concepten rond navigatie binnen een app bespreken.

7.2 Onderdelen van een navigatie

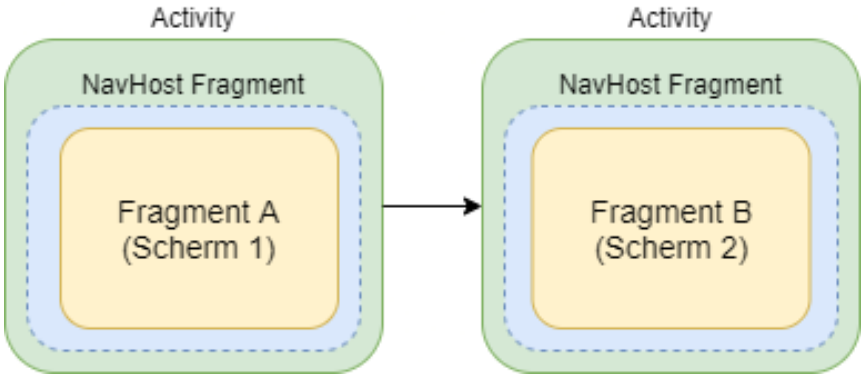
Native Android en Flutter hebben elk een eigen manier van navigeren. Hiervoor doen deze twee systemen beroep op een aantal specifieke componenten. Om navigatie te kunnen verwerken in een native Android app, wordt er tegenwoordig gebruik gemaakt van een component uit de Android Jetpack library. De Navigation Component biedt ontwikkelaars de tools om de navigatie van hun Android apps te implementeren.

Deze component bestaat uit een aantal onderdelen:

- NavHost Fragments
- NavigationControllers
- navigatie xml-bestanden

De user interface waar de navigatie in plaatsvindt, zal een *NavHost* Fragment bevatten. De inhoud van deze NavHost komt overeen met de verschillende schermen van een navigatie, zoals te zien in figuur 7.1. Om de effectieve navigatie op te kunnen roepen, heeft

elke NavHost een *NavController*. Dit is een object waarmee de applicatiecode kan interageren om de navigatie te laten plaatsvinden. Hiervoor biedt deze klasse de *navigate()* en *navigateUp()* methodes aan. Ten slotte maakt de Navigation Component nog gebruik van verschillende xml-bestanden. Elke NavHost krijgt immers een xml-bestand toegewezen, waarin een graaf van navigatie bestemmingen wordt gedefinieerd. Bij deze definitie kan de transitie-animatie opgeven worden voor de overgang tussen de schermen.



Figuur 7.1: Een NavHost voorziet plaats voor de uitwisselbare schermen. (Bracke, 2020)

Een voordeel van het definiëren van de navigatie via deze xml-bestanden is het feit dat deze gemakkelijk visueel voor te stellen is. Het maken van meer complexe structuren is hierdoor een stuk eenvoudiger. De algehele structuur van de Navigation Component is dan weer een nadeel. Om een navigatie op te zetten, zijn er (naast het minimum van twee schermen voor een start en een einde) drie componenten nodig per afzonderlijke navigatie structuur. De NavHost moet in de layouts gezet worden, de xml-definitie moet worden uitgeschreven én de NavController moet worden aangesproken. Wat er ook in het nadeel spreekt voor deze component is het doorgeven van argumenten tussen twee bestemmingen. Deze argumenten worden expliciet meegegeven in de definitie van de betrokken bestemmingen. Hier komt nog eens bij dat, indien type-safety gewenst wordt voor de argumenten, de *SafeArgs* plugin moet worden ingeschakeld. De component heeft ook geen evidente manier om tijdens *navigateUp()* argumenten van de eindbestemming terug te geven naar het oorspronkelijke scherm.

De navigatie binnen Flutter apps is in vergelijking met native Android, een heel stuk eenvoudiger. Flutter maakt voor de navigatie binnen schermen gebruik van de *Navigator* widget. Deze Widget biedt via zijn State object toegang tot de *push()* en *pop()* methodes.

Het eerste verschil dat hier direct opvalt is de eenvoudigere structuur van de navigatie zelf. Er is enkel nood aan een Navigator. In het geval van apps die gebruik maken van de *MaterialApp* of *CupertinoApp* widgets, zit er zelfs al een Navigator in de widget boom. Voor een applicatie met één Navigator is er geen extra setup meer nodig.

Ook pakt Flutter het idee rond argumenten doorgeven anders aan. Bij het opstarten van een navigatie *naar* een bestemming, kan er rechtstreeks een argument worden doorgegeven tijdens de creatie van de widget die de bestemming voorstelt. In de andere richting kan er naar Futures worden gekeken. Aangezien het opstarten van een navigatie in Flutter een asynchrone bewerking is (deze wordt pas 'voltooid' bij het terugkeren naar het oor-

spronkelijke scherm), kan op deze bewerking gewacht worden. Indien er bij het oproepen van *pop()* een argument werd meegegeven, wordt dit het resultaat van de af te wachten Future.

Om de overgangen tussen de schermen te animeren wordt er standaard gebruik gemaakt van de *MaterialPageRoute*[1] bij het oproepen van de navigatie. Dit is een *Route* die voor zowel Android als IOS apps de standaard transitie van de platform specifieke stijlgids zal gebruiken.

Codevoorbeeld 7.1: Navigatie in Flutter met de Navigator. De MaterialPageRoute voorziet een animatie en er worden in beide richtingen argumenten doorgegeven. (Bracke, 2020)

```
Navigator.of(context).push(
  MaterialPageRoute(
    builder: (context) => OtherPage(destinationValue)
  )
).then((returnedValue) => foo(returnedValue))

//Teruggeven van argument, ergens in OtherPage
Navigator.of(context).pop(returnedValue)
```

Om het implementeren van deze verschillende onderdelen in perspectief te zetten zal er worden gekeken hoeveel lijnen code elke component in beslag neemt.

7.3 Navigatie met één pad

Een navigatie opzetten die maar één concreet pad definieert, met enkel een startscherm en een eindbestemming, kan in een native Android app worden opgezet met 27 lijnen code. Dit omvat 11 lijnen voor het definiëren van het NavHost Fragment in een layout bestand. Het opzetten van een bestand voor de navigatie definitie kost 5 lijnen code. Het opzetten van een verbinding tussen een startscherm en een bestemming, kost 10 lijnen code. Ten slotte moet de NavController worden gevraagd om de effectieve navigatie uit te voeren. Dit kost één lijn code.

In vergelijking met de Android app, heeft een Flutter app die vertrekt van een *MaterialApp* widget, enkel nood aan het oproepen van de meegeleverde Navigator. Dit kost 3 lijnen code.

7.4 Navigatie met meerdere paden

Om een wat meer complexere navigatie te vergelijken, zal er worden gekeken naar de opzet van een geneste navigatiestructuur, zoals aanwezig in het keuzescherm van de testap-

[1] De naamgeving van deze klasse is wel wat misleidend. *Material*PageRoute klinkt alsof het enkel slaat op de Android componenten, terwijl dit hier net niet het geval is.

plicatie.

Hiervoor wordt in de native Android app beroep gedaan op twee NavHosts en twee navigatie definities. Het opzetten van elke NavHost/navigatie definitie is op vlak van resultaten gelijk aan deze van de enkelvoudige NavHost. In de praktijk komt dit neer op het tweemaal opzetten van enkelvoudige navigatie, waar de verschillende NavHosts in andere layouts zitten. Om bij een geneste navigatie de *juiste* NavHost te laten navigeren, kan de NavController recursief worden gezocht vanaf een ander startpunt. Eenmaal de juiste NavHost werd gevonden kan een navigatie worden opgeroepen. Indien een genest Fragment (bijvoorbeeld van het keuzescherm uit de testapplicatie) nood heeft aan een NavHost die in de Activity zit, kan deze via één extra lijn code worden opgehaald. In andere gevallen zal er via een interface worden gewerkt. Het uitschrijven van een dergelijke interface zal hier niet worden besproken, aangezien er enkel werd gekeken naar de opstelling van de testapplicatie. De structuur binnen de testapplicatie kan als volgt omschreven worden:

- een NavHost in de Activity
- een NavHost in het keuzescherm
- elke NavHost heeft een start en eindscherm

Voor de resultaten zal er enkel naar de vier betrokken schermen worden gekeken. Andere schermen worden niet meegeteld in de resultaten.

Indien het opzetten van de navigatie voor deze opstelling wordt geïmplementeerd, komt dit neer op een totaal van 52 lijnen code. Het opzetten van de twee NavHosts eist 22 lijnen op van het totaal. De twee grafen voor de navigatie definities eisen de overige 30 lijnen code op.

Het aanroepen van navigatie binnen hetzelfde niveau van de NavHost, kost één lijn code. Het aanroepen van de NavHost op het niveau van de Activity kost 2 lijnen code.

In Flutter apps met een geneste Navigator, is er een stuk extra code nodig om dit soort structuur op te zetten. Ten eerste zal een Navigator in de widget boom gedefinieerd worden. Vervolgens zal deze een *GlobalKey* krijgen, waarmee deze Navigator op te roepen is. Deze sleutel kan in een InheritedWidget worden bewaart of via Dependency Injection worden aangeleverd.

Het opzetten van een extra Navigator kost 11 lijnen code plus één lijn code per extra navigatieroute. Een InheritedWidget, die twee Navigator sleutels zal aanbieden, kost 14 lijnen code om te implementeren. Het ophalen van de juiste Navigator via de sleutels kost één lijn code. Het aanspreken van deze Navigators is identiek aan de niet geneste structuur uit sectie 7.3. Dit kost ook 3 lijnen code voor een navigatie met een specifieke Navigator. De uitgeschreven InheritedWidget moet boven de *MaterialApp* in de widget boom terecht komen, dit kost 4 lijnen code. De sleutel van de Navigator voor de MaterialApp moet ook worden meegegeven, dit kost één lijn code. Alle componenten samen komt dit neer op een totaal van 34 lijnen code.

7.5 Navigatie met argumenten

Om tijdens de navigatie binnen de Android app, een argument mee te kunnen geven via *SafeArgs*, wordt dit argument eerst gedefinieerd in de nodige navigatie definities. Een argument definiëren in een bestemming kost 4 lijnen code. Het meegeven van dit argument aan de NavController kost twee lijnen code. Het argument ophalen in de bestemming kost dankzij de *by navArgs()* Kotlin extensie maar één lijn code.

Codevoorbeeld 7.2: Doorgeven van argumenten via de klasse die werd gegenereerd door SafeArgs. (Bracke, 2020)

```
//In het startende scherm
private fun onNavigateWithArgs(){
    val action = NavigationFragmentDirections
        .actionNavigationFragmentToNavArgsFragment(
            "This arg got passed along!")
    findNavController().navigate(action)
}

//In de bestemming
private val args: NavArgsFragmentArgs by navArgs()
```

In Flutter apps kost het meegeven van een argument tijdens een navigatie naar een bestemming één lijn code. Er moet enkel een argument meegegeven worden tijdens de creatie van de widget. Ook het teruggeven van een argument uit een bestemming kost één lijn code, aangezien dit simpelweg een argument is voor *NavigatorState.pop(argument)*. Zoals eerder vermeld is dit de uiteindelijke waarde van de Future die *push()* opstart.

7.6 Navigatie met animatie

Om in een native Android app een overgang tussen twee schermen te voorzien van een animatie, wordt deze overgang geconfigureerd in de definitie van de navigatiebestemmingen. Dit kost 4 lijnen code per scherm. Er zijn namelijk 4 verschillende animaties die moeten worden opgegeven.

Codevoorbeeld 7.3: Overgangen animeren in een Android app kan via de eigenschappen van de transitie ingesteld worden. („Animate transitions between destinations”, 2019)

```
<fragment
    android:id="@+id/FragmentA"
        ...

        <action
        android:id="@+id/action_FragmentA_to_FragmentB"
        app:destination="@id/FragmentB"
        app:enterAnim="@anim/slide_in_right"
        app:exitAnim="@anim/slide_out_left"
        app:popEnterAnim="@anim/slide_in_left"
        app:popExitAnim="@anim/slide_out_right" />

        ...
</fragment>
```

Het toevoegen van animaties bij Flutter apps kost één lijn code voor elke oproep van een navigatie. Dit omvat enkel het gebruik van een *MaterialPageRoute* als argument voor de

navigatie functie.

7.7 Conclusies uit dit hoofdstuk

Op vlak van navigatie heeft Flutter duidelijk het betere systeem. De algehele structuur is eenvoudiger, er zijn minder componenten nodig in vergelijking met native Android. Flutter heeft ook geen nood aan een plugin om argumenten op een typesafe manier door te geven. Wat ook in het voordeel speelt van Flutter is de eenvoudige manier om argumenten in beide richtingen door te geven. Ten slotte zijn de standaard transitie-animaties out-of-the-box beschikbaar via de MaterialPageRoute klasse.

8. Internationalisering

8.1 Inleiding

Mobiele applicaties hebben meestal een brede doelgroep. Binnen deze doelgroep is de kans groot dat verschillende mensen niet dezelfde taal spreken. Daarom is het noodzakelijk dat een applicatie in verschillende talen aangeboden kan worden.

Hiervoor wordt gebruik gemaakt van een proces genaamd *localization*. Schäler definieert dit proces als het aanpassen van digitale content, op vlak van taal en cultuur, om tegemoet te komen aan de vereisten van een buitenlandse markt. (Schäler, 2010)

In dit hoofdstuk zal er worden gekeken naar een onderdeel van localization, opstellen van vertalingen.

8.2 Vertalen in native Android en Flutter

Native Android apps maken voor het aanbieden van vertalingen gebruik van een reeks xml-bestanden. Deze bestanden bevatten de verschillende vertaalde stukken tekst. Op elk van deze bestanden wordt een taalbeperking gelegd. Het resourcesysteem van Android zal dan via de ingestelde taal bepalen welke bestanden gebruikt moeten worden om de vertaling aan te bieden. Dit systeem heeft als voordeel dat het vertalen van Android apps heel gemakkelijk maakt. Ook het toevoegen van nieuwe talen is hier heel gemakkelijk.

Een Flutter app vertalen kost wel meer werk in vergelijking met native Android. Om een applicatie te configureren voor verschillende talen moeten eerst de ondersteunde talen

expliciet worden opgegeven. Dit kost per taal één lijn code.

Flutter apps maken gebruik van de *LocalizationsDelegate* klasse om de vertalingen aan te bieden. Ontwikkelaars zullen naast hun eigen implementatie van de delegate, ook de drie reeds bestaande delegates moeten meegeven. Een delegate voor Android, één voor IOS en een algemene delegate. Dit kost 4 lijnen code. De delegate zal bepalen welke tekst er doorheen de applicatie wordt gebruikt.

Het uitschrijven van de delegate omvat het maken van eigenschappen in de klasse voor elk vertaald stuk tekst. Ook moeten er een aantal methodes worden geïmplementeerd om de implementatie van de delegate te voltooien. Een minimale delegate implementeren kost 6 lijnen code. Meestal gaat deze klasse gepaard met een extra klasse die de effectieve vertalingen bevat. Zonder te kijken naar een volledige implementatie van dit systeem, kan direct besloten worden dat dit een moeilijkere[1] manier in vergelijking met native Android. Developers moeten immers zelf heel wat code schrijven vooraleer ze effectief kunnen werken met vertalingen. Ook het toevoegen van een nieuwe taal wordt een moeilijke taak. Per taal moet de klasse die de tekst bevat worden uitgebreid en/of overgeërfd.

Het ophalen van een vertaald stuk tekst kan in beide systemen met één lijn code.

8.3 Conclusies uit dit hoofdstuk

Uit de bevindingen van dit hoofdstuk is duidelijk te zien dat native Android een veel betere manier van vertalen aanbiedt. Het resource-systeem van native Android laat toe om de vertalingen op een gemakkelijke manier te gebruiken.

Flutter kan op dit vlak nog veel verbeteren. De implementatie met behulp van de delegates mag dan wel werken, de opzet van deze klassen blijft wel een knelpunt.

[1] Er bestaan wel plugins die dit proces automatiseren via code generatie én het integreren met de bestaande tools van IDE's.

9. Permissies

9.1 Inleiding

Als mens is het belangrijk dat onze privacy wordt gerespecteerd. Dit geldt ook voor mobiele applicaties die we gebruiken in het dagelijkse leven. In dit hoofdstuk zal er worden gekeken hoe de privacy van een gebruiker kan beschermd worden binnen een mobiele app.

9.2 Wat zijn Permissies

Om de privacy van gebruikers te kunnen garanderen maken mobiele applicaties gebruik van applicatie permissies. Dit houdt in dat er voor bepaalde functionaliteit, die betrekking heeft op een deel van de privacy van een gebruiker, toestemming wordt gevraagd. Merk op dat er geen permissie moet worden gevraagd voor de *impliciete* permissies. De impliciete permissies brengen de privacy van de gebruiker niet in het gedrang. Toegang krijgen tot Wifi of een dataverbinding, is een voorbeeld van een impliciete permissie.

9.3 Permissies in native Android en Flutter

Native Android maakt gebruik van een specifieke structuur om een permissie op te vragen. Voor een stuk functionaliteit dat een bepaalde permissie nodig heeft, zal er eerst worden gekeken of de permissie al werd goedgekeurd door de gebruiker. In dit geval kan de applicatie de instructies uitvoeren. Indien dit niet het geval is, zal het systeem

een melding tonen met een omschrijving specifiek aan de permissie die wordt verwacht. De gebruiker kan via deze melding toestemming geven of de permissie weigeren. Als de gebruiker de permissie heeft goedgekeurd of geweigerd zal de applicatie een resultaat ontvangen. Indien uit het resultaat blijkt dat er toestemming werd gegeven, kan de applicatie de functionaliteit uitvoeren. Indien de permissie werd geweigerd, zal de applicatie de functionaliteit moeten uitschakelen.

Codevoorbeeld 9.1: Er kan aan de gebruiker om toestemming worden gevraagd, indien een functionaliteit betrekking heeft op de privacy. („Request App Permissions", 2019)

```
// Voorbeeld met de Camera permissie
if (ContextCompat.checkSelfPermission(thisActivity,
        Manifest.permission.CAMERA)
        != PackageManager.PERMISSION_GRANTED) {

    if (ActivityCompat.shouldShowRequestPermissionRationale(
                thisActivity, Manifest.permission.CAMERA)) {
        //Toon een medling die verklaart
                //waarom de permissie nodig is
    } else {
        // Vraag Toestemming
        ActivityCompat.requestPermissions(thisActivity,
                arrayOf(Manifest.permission.CAMERA),
                MY_PERMISSIONS_CAMERA)
    }
} else {
    // App heeft reeds toestemming
}
```

Het uitschrijven van een implementatie waarin om één permissie wordt gevraagd, zoals hierboven beschreven, kost 21 lijnen code. Dit omvat 9 lijnen code om de permissie op te vragen via het tonen van een melding. Het opvangen van de toestemming kost 11 lijnen code. Ten slotte moet elke permissie in het manifest van de app worden vermeld, dit kost één lijn code per permissie.

Flutter biedt voor het opvragen van toestemming via permissies zelf geen functionaliteit aan. Dit is te verklaren door het feit dat dit altijd langs het native platform zal moeten passeren. Om toch een permissie te kunnen vragen zijn er drie mogelijkheden om via het native systeem een permissie op te vragen.

De eerste mogelijkheid is het uitschrijven van een Method Channel. Via deze Method Channel zal het native platform worden aangesproken, waar het permissie systeem van native Android (of een ander platform) de permissie zal opvragen. Method Channels zullen verder worden besproken in hoofdstuk 13. Deze implementatie kost uiteraard meer code in vergelijking met native Android. Naast de verplichte code voor het opvragen van de permissie via het native systeem (zoals hierboven beschreven) moet er nog een Method Channel worden uitgeschreven. Anders is er geen verbinding tussen het opvragen/ontvangen van de permissie en de Flutter app.

Om zelf geen Method Channel te moeten uitwerken, kan er gebruik worden gemaakt van een plugin die instaat voor het aanspreken van de native code. Hier kan bijvoorbeeld de *permission_handler* plugin worden gebruikt. Dit heeft twee voordelen. Een ontwikkelaar hoeft zich hierdoor niet bezig te houden met het uitwerken van native code. Ook kan hier de cross-platform aard van Flutter worden ingezet. Dankzij de plugin zal er een native implementatie zijn voor de verschillende platformen én wordt de implementatie verborgen via een API die via Dart kan worden aangesproken. De keuze van een plugin is wel iets waar over nagedacht moet worden, niet alle plugins zijn mee met de huidige stand van Flutter en/of de native platformen.

Het opvragen van één permissie via deze plugin, kan worden geïmplementeerd met 1 lijn code. Het nagaan van de toestemming kan via deze plugin worden gerealiseerd met 4 lijnen code (inclusief het opvragen van de permissie). Ook hier moet de permissie in het manifest worden gezet. Een volledige implementatie kost 5 lijnen code.

Indien de permissies te maken hebben met een héél specifiek stuk functionaliteit (bijvoorbeeld het selecteren van een afbeelding uit de galerij van een gebruiker), kunnen ontwikkelaars via een plugin de permissies bundelen met de native code voor de functionaliteit die oorspronkelijk de permissie nodig had.

Plugins die een bepaalde native functionaliteit beschikbaar maken voor Flutter apps, moeten immers altijd de permissie vragen om hun uiteindelijke doel te bereiken. Dankzij dit soort plugins hoeven ontwikkelaars zelfs helemaal geen code meer te schrijven in verband met permissies. De API van de plugin zal op basis van de toestemming een andere functionaliteit laten doorgaan.

9.4 Conclusies uit dit hoofdstuk

Flutter biedt voor het afhandelen van permissies de gemakkelijkste werkwijze. Dankzij het bestaande plugin systeem kan de nodige native code worden verborgen achter een Dart API. Deze werkwijze maakt het werken met permissies iets gemakkelijker, op voorwaarde dat de gebruikte plugins up to date zijn. Enkel in de situatie waar er geen plugin bestaat voor de functionaliteit, is native Android de betere keuze.

10. Software Testen

10.1 Inleiding

Naast het uitschrijven van een implementatie van een applicatie, is het ook belangrijk dat het gedrag van de applicatie kan gevalideerd worden. Hiervoor worden software testen geschreven. Dit hoofdstuk zal een aantal verschillende types van deze software testen bespreken.

10.2 Unit Testen

De eerste soort software test die kan geschreven worden is de *Unit Test*. Een Unit Test wordt gebruikt om een klein deel van de business logica van een applicatie te valideren. Unit Tests maken gebruik van het AAA patroon. AAA staat voor *Arrange, Act and Assert* en is de werkwijze waarmee een test wordt geschreven. Eerst wordt de nodige configuratie opgezet in de *Arrange* stap. Daarna volgt de *Act* stap. In deze stap wordt de code die in deze Unit Test zal geëvalueerd worden, eenmaal uitgevoerd. Ten slotte wordt het resultaat gevalideerd in de *Assert* stap.

Een Unit Test opzetten in Flutter kost één lijn code. De effectieve implementatie van de testcode via de AAA methodiek is hier minder relevant, aangezien deze per test anders is. Elke test moet geschreven worden binnen een *main()* methode, dit is het startpunt van een Dart applicatie. Dit staat los van Flutter zelf. Deze methode kost ook één lijn code per collectie van testen (een enkele test óf een *group()* van meerdere testen).

Een Unit Test in Android kost twee lijnen code. De signatuur van de testmethode kost

hiervan één lijn code en de *@Test* annotatie eist ook een lijn op.

10.3 User Interface Testen

Naast het testen van de business logica is het ook belangrijk dat het gedrag van een user interface getest kan worden.

Android biedt hiervoor de Espresso library aan. Dit is een toolkit die toelaat om tijdens de Act en Arrange stappen met de user interface te werken. (Lämsä, 2017)

Flutter biedt voor het testen van de Widgets binnen de user interface, de zogenaamde *WidgetTester* aan. Dit is een adaptatie van de werkwijze rond Unit Tests, die op een asynchrone manier met Widgets werkt.

Een UI test uitschrijven in Flutter kost net zoals de Unit Tests één lijn code (exclusief de AAA implementatie). Ook hier is één lijn code nodig voor de opstart via de main methode. De WidgetTester heeft wel een nadeel, deze roept namelijk zelf niet de *setState()* methodes van de StatefulWidgets aan. Hiervoor moet handmatig *pumpAndSettle()* worden opgeroepen, voor elk moment waarop een Widget moet veranderen binnen de user interface. Bij dit soort testen komt wel het grote voordeel van het Widget systeem naar boven. Aangezien Widgets modulair zijn, kunnen deze op een eenvoudige manier apart getest worden. Een 'testversie' opzetten houdt in dit geval in dat de WidgetTester met een andere Widget boom wordt gestart. Hier kan ook via Dependency Injection gewerkt worden indien nodig.

Codevoorbeeld 10.1: Een Widget test in Flutter met behulp van de WidgetTester. (Bracke, 2020)

```
testWidgets('Sample UI Test', (WidgetTester t) async {
    // Build our app and trigger a frame.
    await t.pumpWidget(MyApp());
    await t.pumpAndSettle();

    //Tap item 2
    await t.tap(find.byIcon(Icons.add).at(1));
    //Do a redraw
    await t.pumpAndSettle();
    //Verify that the title changed
    expect(find.text("two"), findsOneWidget);
});
```

Een UI test opzetten in Android is, in vergelijking met Flutter iets moeilijker. De testmethode moet worden gemarkeerd als een UI test, dit kost drie lijnen code (inclusief de declaratie van de test zelf). Het opzetten van een testversie van de applicatie kost minimum één lijn code. Er moet ook een extra klasse worden geschreven die de *testversie* van de app zal inladen tijdens de test. Dit kost 4 lijnen code. Als laatste stap moet er een *ActivityScenario* of een *FragmentScenario* worden opgestart. Dit kost één lijn code. Hierna

kan de AAA implementatie worden uitgeschreven. Een UI test opzetten kost 9 lijnen code. In vergelijking met Flutter, is deze werkwijze niet zo modulair. Om bijvoorbeeld een simpel stuk tekst te testen, is minimum een volledig Fragment mét een TextView nodig. Deze test zal altijd een FragmentScenario nodig hebben. In Flutter zou er enkel een Text widget meegegeven kunnen worden aan de WidgetTester.

10.4 Integratie Testen

Het laatste type test dat hier behandeld zal worden is de Integration Test.

In Android verschillen de Integration testen niet zoveel van de UI testen. Een Integration test in Android is een UI test die meer stappen doorloopt in de applicatie. Integration testen beperken zich niet tot één aspect van de user interface maar doorlopen een volledig proces. Bijvoorbeeld: Iemand wil een product aankopen. Hier zal een Integration Test alle stappen doorlopen, van het kiezen van het product tot de effectieve betaling.

Hier kunnen de resultaten van de UI testen worden hergebruikt, aangezien de AAA implementatie specifiek is aan de test.

Een Integration Test uitschrijven in Flutter gebeurt via de *Flutter Driver*. De Flutter Driver maakt het mogelijk om via een fysiek toestel (of een emulator) een test uit te voeren. Waar de Widget Testen niet op een toestel hoeven te draaien, is dit bij de Driver wel het geval. Een opstelling van een Integration Test is in vergelijking met Android wel wat ingewikkelder. De Driver verwacht een geïnstrumenteerde versie van een applicatie, dit kost 3 lijnen code. Naast het uitschrijven van de geïnstrumenteerde versie, moet de test zelf nog worden opgesteld. De testcode is qua lijnen code ongeveer gelijk aan die van een Widget Test. Bij een Integration Test komt er wel nog een stuk opstelling bij voor de Driver zelf. Dit kost 5 lijnen code. Als hier de setup van een Widget Test wordt bijgeteld komt het totaal voor een Integration Test op 10 lijnen code te staan.

Codevoorbeeld 10.2: Espresso maakt het mogelijk om de user interface van een Android app te testen. Er kunnen bijvoorbeeld Integratietesten worden uitgewerkt. (Bracke, 2020)

```
//Launch an activity Scenario that starts MainActivity
ActivityScenario.launch(MainActivity::class.java)
//Main Activity is now on screen, do the test
//Tap Menu
openActionBarOverflowOrOptionsMenu(ctx)
//Tap Item 3
onView(withId(R.id.three)).perform(click())
//find title text, should be updated to the new string
onView(
    allOf(
        instanceOf(TextView::class.java),
        withParent(withId(R.id.toolbar))
    )
).check(matches(withText(app.getString(R.string.three))))
```

10.5 Conclusies uit dit hoofdstuk

Uit de resultaten van dit hoofdstuk blijkt dat beide systemen equivalent zijn bij het uitschrijven van Unit Tests. Indien individuele stukken van een user interface onderworpen worden aan een test, is Flutter de betere kandidaat. De modulariteit van het Widget systeem is ook tijdens het schrijven van Widget Tests een pluspunt. Bij het uitschrijven van Integration Tests biedt native Android de betere optie, dankzij de minimale setup kost.

11. Opstart Performantie

11.1 Inleiding

Gebruikers van software applicaties vinden de gebruikerservaring heel belangrijk. Een applicatie waar de processen die een gebruiker kan doorlopen eenvoudig zijn, biedt een betere gebruikerservaring voor de eindgebruiker. Ook de opstarttijd van een applicatie heeft een effect op de gebruikerservaring. Een snelle applicatie zal immers veel meer gebruikers aan trekken. In dit hoofdstuk zullen zowel Android als Flutter geëvalueerd worden op basis van de opstarttijd.

11.2 Opstelling

Om de opstarttijd op een consistente manier te kunnen meten werd voor beide systemen een minimale applicatie geschreven. De minimale applicatie volgt de *HelloWorld* template. Dit is een applicatie met maar één scherm. Dit scherm bevat enkel de tekst *HelloWorld!*, die zich in het midden bevindt. Het tekenen van het eerste *frame* kan als meetstaaf gebruikt worden. Dit is het moment waarop de gebruiker de user interface van de applicatie voor de eerste keer ziet.

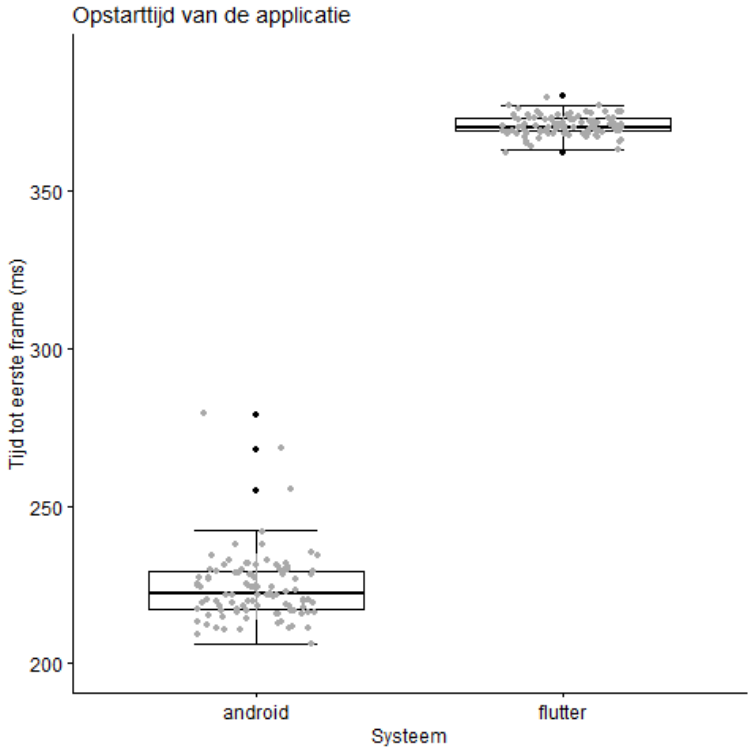
Omdat de performantie niet gehinderd mag worden door externe factoren, wordt de applicatie getest zonder een debugger. De debugger is een tool die wordt gebruikt tijdens het ontwikkelen en biedt een aantal extra's aan, ten koste van performantie. Het uitvoeren van performantietesten is in Android te realiseren via de *release* modus. Flutter biedt een

extra modus aan, specifiek voor het meten van de performantie. Deze *profile*[1] modus is perfect geschikt om de performantie van de Flutter app te meten.

De opstarttijd van de applicaties zal worden gemeten via het uitvoeren van een *Cold Start*. Een Cold Start houdt in dat de applicatie volledig van nul wordt opgestart.

11.3 Resultaten

Na het uitvoeren van 100 performantietesten voor beide applicaties kan er worden besloten dat de native Android app veel sneller opstart in vergelijking met de Flutter app. Zoals te zien in figuur 11.1 kan een minimale Android app worden opgestart in gemiddeld 224 ms. Hier is ook uit te concluderen dat de resultaten van de Android app wel wat uit elkaar liggen. De standaardafwijking bedraagt immers 10,9 ms.



Figuur 11.1: De resultaten van de honderd metingen voor de opstarttijd. De zwarte punten zijn hier de uitschieters. (Bracke, 2020)

In vergelijking met Android heeft de Flutter app wel wat extra tijd nodig totdat het eerste frame op het scherm komt. Het gemiddelde ligt hier aanzienlijk hoger, op 370,7 ms. De opstarttijd van de Flutter applicatie is wel een stuk stabieler. De standaardafwijking is hier maar een magere 3 ms.

[1]De profile modus van het Flutter framework breidt de *release* modus uit met een aanhakingspunt voor de Dart DevTools.

Waarom heeft Flutter meer tijd nodig om op te starten? Dit is te verklaren door de aanwezigheid van de *FlutterEngine*, zoals reeds vermeld in hoofdstuk 4. De Flutter Engine heeft immers ook tijd nodig om zichzelf klaar te maken. Vandaar dat Flutter apps iets meer tijd nodig hebben.

11.3.1 Extra Opmerking over de resultaten

De performantietesten in dit onderzoek maken gebruik van versie 1.12 van het Flutter framework. In de volgende grote release, versie 1.17, zijn een aantal zaken verbeterd. Hieronder vallen ook de opstarttijd en de geheugenafdruk van Flutter apps. (Sells, 2020)

12. Applicatiegrootte

12.1 Inleiding

Naast de algemene gebruikerservaring is er nog een belangrijk punt waarop applicaties goed moeten scoren, de effectieve grootte van de app. De opslag van het toestel van een gebruiker is immers beperkt. Gebruikers hebben ook een hele verzameling applicaties op hun toestel staan. Indien elk van deze apps veel plaats innemen op het opslagmedium, zal de gebruiker niet veel apps kunnen installeren. In dit hoofdstuk zal er worden gekeken naar de grootte van Android en Flutter apps.

12.2 Opzet

Om de grootte van de applicaties te meten zal er worden gekeken naar de bestandsgrootte van het uitvoerbare bestand van de applicatie. Dit bestand staat ook bekend als de *executable*. Om gen last te hebben van externe factoren zullen de executables van beide applicaties worden gemaakt via de *release* modus. De release modus is een versie van een applicatie die geschikt is om op de respectievelijke App Store te zetten. Deze versies worden zo klein mogelijk gemaakt zodat deze een kleiner aandeel op eisen van de totale opslag. Hiervoor wordt de *minifyEnabled* optie aangezet. De Flutter template doet dit reeds voor het *release* build type[1]. Net zoals in het hoofdstuk 11 wordt hier de Hello-World template gebruikt. Zo kan een basislijn getrokken worden voor de uiteindelijke executable.

[1] Dit is reeds gedefinieerd in het *flutter.gradle* bestand.

Tabel 12.1: De uiteindelijke grootte van de Hello World apps. (Bracke, 2020)

Android	Flutter
1.99 MB	5.4 MB

12.3 Resultaten

De executable van een Android app wordt ook wel een Android Package (APK) genoemd. Deze APK bevat een aantal *.dex* bestanden die door de Android Runtime (ART)[2] uitgelezen kunnen worden (Ehringer, 2010). ART zal de instructies in deze .dex bestanden uitvoeren wanneer de applicatie draait.

Zoals tabel 12.1 illustreert is de APK van de Android app zeer klein. Deze is met zijn 1.99 MB net geen 2 megabyte groot. In vergelijking met Android, is de Flutter app iets groter. Deze is met zijn 5,4 megabyte duidelijk een pak groter dan de Android APK. De oorzaak van dit verschil kan net zoals in hoofdstuk 11 toegewezen worden aan de aanwezigheid van de Flutter Engine. Deze wordt mee verpakt in de APK. Zonder de Flutter Engine zou de Flutter app immers niet kunnen functioneren.

[2]Dit is de opvolger van de Dalvik Virtual Machine.

13. Appendix: Method Channels

13.1 Inleiding

Aangezien Flutter een Cross-Platform framework is, kan er zich een situatie voordoen waarbij er een bepaald stuk functionaliteit nodig is, dat niet via Dart code uit te werken is. Om dergelijke functionaliteit toch te kunnen implementeren biedt Flutter een extra feature aan, de Method Channels. Dit hoofdstuk zal kort de werking van deze Method Channels bespreken.

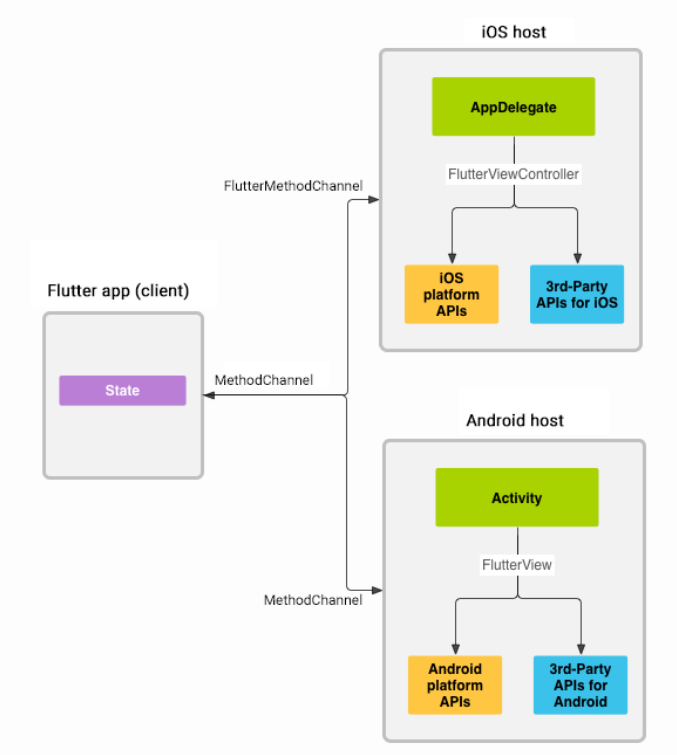
13.2 Wat is een Method Channel

Alhoewel het Flutter framework zelf een aantal functionaliteiten biedt, is er soms nog nood aan het aanspreken van het native platform. Het native platform biedt namelijk API's aan die met de hardware van de toestellen van gebruikers communiceert.

Indien Flutter zelf geen equivalente implementatie voorziet, zal het toch nodig zijn om deze API's aan te spreken. Om via de Flutter kant van een applicatie met de native API's te kunnen werken, kan een Method Channel worden opgezet.

Een Method Channel maakt gebruik van het *Message Passing* patroon. Dit is een manier om twee systemen met elkaar te verbinden, zonder dat deze veel van elkaar weten. De communicatie verloopt doormiddel van het uitsturen van berichten van systeem A naar systeem B en omgekeerd. (Barker, 2015)

In het Dart deel van de applicatie wordt een Method Channel geregistreerd. In de native



Figuur 13.1: De Method Channels in elk van de native platformen zijn in staat om te communiceren met het Dart gedeelte van een applicatie. (Thomsen, 2017)

platformen (Activity/AppDelegate) wordt ook een Method Channel opgezet. Zo ontstaat er een brug die in beide richtingen kan werken. Een dergelijke opstelling is te zien in figuur 13.1.

Om de berichten door te kunnen sturen, maakt een Method Channel gebruik van een *Codec*. Deze codec staat in voor het encrypteren en decrypteren van de berichten tijdens het verzenden en ontvangen. Flutter biedt al een aantal implementaties van de Codec aan. Deze variëren van de *BinaryCodec* tot de *JSONMessageCodec*.

14. Conclusie

Flutter mag dan wel een jong framework zijn, uit de resultaten van dit onderzoek is wel gebleken dat er voor alle onderzochte onderdelen een implementatie bestaat.

Uit de bevindingen van hoofdstuk 4 blijkt dat het ontwikkelen van user interfaces met behulp van het Flutter framework ongeveer evenveel voor- en nadelen heeft ten opzichte van native development voor het Android platform. Beide systemen overtreffen elkaar op de helft van de besproken onderdelen binnen dit hoofdstuk.

In hoofdstuk 5 wordt native Android benadeeld door de minpunten van de beschikbare API's. Flutter biedt hier ook een aantal extra's aan, die native Android niet ter beschikking stelt. Het is duidelijk dat Flutter hier de betere keuze is.

De resultaten van hoofdstuk 6 tonen aan dat Flutter een bredere waaier aan opties biedt voor het uitwerken van een oplossing voor lokale persistentie. Android hangt wat vast aan de reeds beschikbare relationele databases, terwijl Flutter wel een aantal niet-relationele databases kan aanbieden.

Hoofdstuk 7 toont aan dat de ontwikkelingsmethode voor het uitwerken van navigatie binnen een app, een heel stuk eenvoudiger is in Flutter. De driedelige structuur van de Navigation Component kan de simpliciteit van de Navigator niet evenaren.

Hoofdstuk 8 concludeert dat native Android de betere kandidaat is bij het uitwerken van vertalingen. Het bestaande resource-systeem kan hier gelukkig veel automatiseren, wat vertalen heel eenvoudig maakt. De bestaande werkwijze voor het uitwerken van vertalingen in Flutter is een groot minpunt voor het framework.

Hoofdstuk 9 bewijst dat Flutter op vlak van permissies zeer flexibel is. Het bestaande

plugin systeem kan deze functionaliteit mooi integreren met de Cross-Platform aard van het framework. Met als gevolg dat het gebruiken van permissies veel gemakkelijker is in vergelijking met native Android.

In hoofdstuk 10 is gebleken dat voor het uitwerken van software testen, beide systemen ongeveer gelijk lopen. Unit Tests zijn voor beide systemen gemakkelijk op te zetten. Een UI test uitwerken is in het Flutter framework iets eenvoudiger, dankzij de koppeling met het Widget systeem. Native Android biedt de betere werkwijze bij het uitwerken van Integration Tests. De Flutter Driver heeft wat extra setup nodig, wat in het nadeel speelt van het framework.

Hoofdstukken 11 en 12 tonen aan dat native Android beter is op vlak van performantie gerelateerde zaken. Zowel de applicatiegrootte als de opstarttijd van de Flutter applicatie is slechter in vergelijking met de native app. Flutter is en blijft een framework, hier zal het native systeem dus altijd beter blijven.

Indien de scores van de eindresultaten worden opgeteld[1], krijgt het Flutter framework een score van 4/7 en heeft native Android recht op een score van 3/7.

De twee hypotheses uit sectie 1.3 kunnen worden ontkracht. Ten slotte kan er een antwoord geformuleerd worden op de onderzoeksvragen uit sectie 1.2:

Voor het ontwikkelen van de onderzochte features is Flutter meestal de efficiëntste optie.

Het framework kan voor alle features uit de native development stack van Android een implementatie aanbieden. Dit wil niet zeggen dat alle features even gebruiksvriendelijk zijn naar de ontwikkelaars toe. De huidige implementatie van bepaalde features kan in de toekomst nog verbeteren. Het framework is nog jong, het is normaal dat sommige features nog nood hebben aan een nóg efficiëntere implementatie.

14.1 Toekomstig Onderzoek

Dit onderzoek kijkt enkel naar de Android kant van native ontwikkeling. Logischerwijs moet native development voor IOS ook onderzocht worden. Als uitgangspunt voor de meeste metrics werd het aantal lijnen code genomen. De ontwikkelingstijd voor een gemiddelde applicatie is ook een goede metric om een onderzoek zoals dit te voeren. Denk aan een tweetal ontwikkelaars die op hetzelfde moment starten met delen van een template app te ontwikkelen, één versie per onderzocht systeem. Per deel kan een meting voor de ontwikkelingstijd worden genomen. De resultaten van de performantie gerelateerde onderdelen van dit onderzoek kunnen ook snel verouderen. Dit is te danken aan het feit dat beide systemen vrij snel veranderen. Het is dus aan te raden om zulke metingen opnieuw te doen, eenmaal er een aantal grote releases zijn gepasseerd.

[1] Hierbij wordt bij een gelijkspel geen punt gegeven. Geen van beide is in dit geval de doorslaggevende optie.

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

Applicaties ontwikkelen voor smartphones, tablets en andere mobiele devices, wordt alsmaar gemakkelijker voor ontwikkelaars. Mede dankzij de tools die zij ter beschikking hebben. Voor deze applicaties te ontwikkelen zijn er twee methoden, native development enerzijds en cross-platform development anderzijds. De eerste, meest voorkomende methode, is het ontwikkelen van een applicatie per doelplatform. De tweede methode koppelt zich los van een specifiek doelplatform en legt de nadruk op de herbruikbaarheid van middelen. Nu is het telkens aan de ontwikkelaars om te kiezen voor de eerste of de tweede manier. De eerste biedt meestal een meer vertrouwde aanpak, aangezien het de norm was nog voor de cross-platform frameworks bestonden. De tweede is echter een stuk efficiënter op development vlak. Er kan namelijk werk worden uitgespaard door bepaalde niet-platform-specifieke elementen te hergebruiken. Dit onderzoek zal zich meer verdiepen in één zo'n cross-platform framework, Flutter. Dit framework zal worden vergeleken met een native Android oplossing, om een beter beeld te geven over hoe het zich verhoudt ten opzichte van native development. Aangezien Flutter zelf relatief jong is, hebben de development teams binnen Endare maar een beperkte ervaring met het framework. Dit maakt het moeilijker om een meer gefundeerde keuze te maken tussen Flutter zelf en andere frameworks. Daarom zal dit onderzoek voor hen een beter beeld proberen schetsen rond de staat van het framework, zodat ze bij de keuze om cross-platform te werken, ook dit framework kunnen overwegen. Er zal een antwoord worden geformuleerd op de

volgende onderzoeksvragen: “Is het ontwikkelen van features met Flutter even efficiënt als in native development?” en “ Kan Flutter tegemoetkomen aan de features van native development?”

A.2 Literatuurstudie

Rond het Flutter framework is, op het moment van schrijven (najaar 2019), weinig research te vinden. Dit valt te verklaren dankzij het feit dat het framework zelf nog zeer jong is. De eerste stabiele release van het framework dateert van december 2018. Veel research waar Flutter onderdeel van was, ging over vergelijkende studies ten opzichte van gelijkaardige frameworks zoals Multi-OS Engine(Fayzullaev, 2018), React Native (Dagne, 2019; Rodríguez-Sánchez Guerra, 2018; Tuusjärvi, 2019; Wu, 2018; Yatsenko e.a., 2019) en PhoneGap/Cordova (Gonsalves, 2019). De research van Dang en Skelton (2016) vergeleek Flutter in een eerder educatieve context. Hierbij werd Flutter vooral uitgespeeld tegenover concurrerende frameworks. Echter werd Flutter nooit ten opzichte van een native development aanpak vergeleken. Dit maakt het moeilijk voor developers om de impact van het framework te meten in vergelijking met native development. Er wordt tot op de dag van vandaag echter nog altijd veel voor native development gekozen(Francese, Gravino, Risi, Scanniello & Tortora, 2017), wat deze vergelijking wel noodzakelijk maakt. Het onderzoek van Gonsalves (2019) ging wel dieper in op een aantal aspecten, waarvan enkele in dit onderzoek opnieuw zullen worden herbekeken. Echter werd in dit onderzoek wel vermeld dat het framework nog in beta stadium was(Gonsalves, 2019, p. 76).

A.3 Methodologie

Er zullen tijdens dit onderzoek twee applicaties worden geschreven. De eerste applicatie zal gebruik maken van de Android Software Development Kit (Android SDK) en zal worden geschreven met behulp van de programmeertaal Kotlin. De tweede applicatie zal gebruik maken van het Flutter framework en wordt geschreven in Dart. Flutter maakt achter de schermen ook gebruik van de Android SDK. Beide applicaties zullen worden ontwikkeld met behulp van de Android Studio Integrated Development Environment. Voor Flutter worden ook de Android Studio Flutter en Dart plugins ingeschakeld. De applicaties zullen onderworpen worden aan de test criteria van dit onderzoek: internationalisering, navigatie, persistentie van gegevens in de applicatie zelf, creëren van een user interface, testing en asynchroon werk. Hiervoor zullen de applicaties een vooropgesteld voorbeeld volgen. Dit template is een concept voor de te schrijven applicatie, waarmee wordt gegarandeerd dat elk van de test criteria aan bod komt.

De executables van beide afgewerkte applicaties zullen worden vergeleken op grootte. Ook zal de opstarttijd van de afgewerkte applicaties worden gemeten. Om het persistentie criterium te testen zal er worden gekeken naar de mogelijkheden om dit te implementeren per applicatie. Indien er meerdere opties beschikbaar zijn, zullen de opties met elkaar worden vergeleken op vlak van flexibiliteit en onderhoudbaarheid. Om het navigatie systeem

van beide applicaties te testen zal er worden gekeken naar de hoeveelheid code die nodig is om een wat complexere navigatie uit te werken. Hiervoor zal een niet geneste navigatie structuur én een eenmaal geneste navigatie structuur worden opgebouwd in beide applicaties. Het ontwikkelen van een user interface, het visuele gedeelde van een applicatie, zal ook worden onderzocht. Hier zal worden gekeken naar het ontwikkelen voor verschillende schermdimensies en oriëntaties. Voor beide applicaties zal worden nagegaan of het mogelijk is om unit tests/ user interface tests en integration tests te schrijven. Unit tests omvatten het testen van een stuk applicatie logica binnen de kleinst mogelijke eenheid, een unit. User interface testen dienen om een visueel gedeelte zoals een scherm te testen. Integration tests testen dan weer een hele flow binnen een app, bijvoorbeeld het aankopen van een artikel in een winkel app. Het werken met asynchrone code zal ook worden vergeleken voor beide apps. Hier zal het opzetten van een Coroutine, een Kotlin term voor een taak die op de achtergrond draait, worden vergeleken met Futures, de manier van asynchroon werken in Flutter. Ten slotte zal er nog gekeken worden naar het implementeren van internationalisering binnen de applicaties. Hiervoor wordt vooral gekeken naar de simpliciteit van het systeem dat de ontwikkelaar ter beschikking wordt gesteld. Beide applicaties zullen worden uitgevoerd op een echt toestel om een nauwkeuriger beeld te kunnen geven dan een emulator.

A.4 Verwachte resultaten

De executable van de native development cyclus wordt wel verwacht de kleinste voetafdruk te hebben. Over de opstarttijd wordt verwacht dat de native applicatie sneller opstart. Dit omdat Flutter een framework is en het ook zijn eigen resources nodig heeft. Op vlak van het software gedeelte wordt er wel van Flutter verwacht dat het de software features die het al aanbiedt, op een meer ontwikkelaars vriendelijke manier aan de man brengt dan de native tegenhanger.

A.5 Verwachte conclusies

Van dit onderzoek wordt verwacht dat voor elk van de onderzochte criteria een duidelijk beeld kan worden geschetst zodat het Flutter framework ofwel aan- of afgeraden kan worden als cross-platform framework. Aangezien het framework zelf snel vooruitgaat, lijkt het wel aannemelijk dat dit advies eerder positief zal worden. De jonge leeftijd van het framework kan waarschijnlijk wel wat nadelen veroorzaken. De kans is namelijk groot dat er nog een aantal dingen niet op punt staan.

Bibliografie

Animate transitions between destinations. (2019). Verkregen 9 mei 2020, van https://developer.android.com/guide/navigation/navigation-animate-transitions

Barker, B. (2015). Message passing interface (mpi). In *Workshop: High Performance Computing on Stampede* (Deel 262).

Belchin, M. & Juberias, P. (2015). Asynchronous Programming with Dart. In *Web Programming with Dart* (pp. 279–297). Springer.

Bracke, N. (2020). Broncode van de applicaties. Verkregen 8 mei 2020, van https://github.com/BrackeNavaron/bachproef_sourcecode

Dagne, L. (2019). Flutter for cross-platform App and SDK development.

Dang, D. & Skelton, D. (2016). Teaching Mobile App Development: Choosing the best development tools in practical labs.

Eck, D. J. (2016). *Introduction to Computer Graphics*. David J. Eck.

Ehringer, D. (2010). The dalvik virtual machine architecture. *Techn. report (March 2010)*, *4*(8).

Fayzullaev, J. (2018). Native-like cross-platform mobile development: Multi-os engine & kotlin native vs flutter.

Fortuna, E. (2019). Keys! What are they good for? Verkregen 1 april 2020, van https://medium.com/flutter/keys-what-are-they-good-for-13cb51742e7d

Francese, R., Gravino, C., Risi, M., Scanniello, G. & Tortora, G. (2017). Mobile app development and management: results from a qualitative investigation. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems* (pp. 133–143). IEEE Press.

Gonsalves, M. (2019). Evaluating the mobile development frameworks Apache Cordova and Flutter and their impact on the development process and application characteristics.

Goranova, M., Kalcheva-Yovkova, E. & Penkov, S. (2015). Task-based Asynchronous Pattern with async and await. In *International Scientific Conference Computer Science* (p. 150).

Lämsä, T. (2017). Comparison of GUI testing tools for Android applications. *University of Oulu*.

Luca Crisan, A. (2019). *A study of Kotlin's: conciseness, safety and interoperability.* (B.S. thesis, Universitat Politècnica de Catalunya).

Otto, M. (2015). Bootstrap 4 Responsive Breakpoints. Verkregen 4 april 2020, van https://getbootstrap.com/docs/4.0/layout/overview/#responsive-breakpoints

Request App Permissions. (2019). Verkregen 9 mei 2020, van https://developer.android.com/training/permissions/requesting

Rodríguez-Sánchez Guerra, M. (2018). Cross-platform development frameworks for the development of hybrid mobile applications: Implementations and comparative analysis.

Schäler, R. (2010). Localization and translation. *Handbook of translation studies*, *1*, 209–214.

Sells, C. (2020). Announcing Flutter 1.17. Verkregen 16 mei 2020, van https://medium.com/flutter/announcing-flutter-1-17-4182d8af7f8e

Sharma, Y. & Gupta, S. (2020). A Study of Flutter and React Native for Mobile App Development. *Our Heritage*, *68*(27), 691–698.

Sproch, J., Powell, A. & Guy, R. (2019). Declarative UI Patterns (Google I/O 2019). Verkregen 29 maart 2020, van https://www.youtube.com/watch?v=VsStyq4Lzxo

Sun, W., Chen, H. & Yu, W. (2017). The Exploration and Practice of MVVM Pattern on Android Platform. In *2016 4th International Conference on Machinery, Materials and Information Technology Applications*. Atlantis Press.

Thomsen, M. (2017). Platform Channels Architecture. Verkregen 29 maart 2020, van https://flutter.dev/docs/development/platform-integration/platform-channels

Tuusjärvi, K. (2019). RestaPoints Mobiilisovelluksen Kehittäminen.

Wu, W. (2018). React Native vs Flutter, Cross-platforms mobile application frameworks.

Yatsenko, R., Obodiak, V. & Yatsenko, V. (2019). COMPARATIVE ANALYSIS OF CROSS-PLATFORM FRAMEWORKS FOR MOBILE APPLICATIONS DEVELOPMENT. *ΛΌΟΣ.*, (4), 132–136.