HO GENT

Faculteit Bedrijf en Organisatie

Cross-platform Flutter ontwikkeling tegenover native Android ontwikkeling: een vergelijkende studie

Sébastien De Pauw

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

Promotor:
Ozgür Akin
Co-promotor:
Navaron Bracke

Instelling: Next Apps BV

Academiejaar: 2020-2021

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Cross-platform Flutter ontwikkeling tegenover native Android ontwikkeling: een vergelijkende studie

Sébastien De Pauw

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

Promotor:
Ozgür Akin
Co-promotor:
Navaron Bracke

Instelling: Next Apps BV

Academiejaar: 2020-2021

Tweede examenperiode

Woord vooraf

Deze bachelorproef werd voltooid in kader van de opleiding Toegepaste Informatica met specialisatie mobiele applicaties. Mijn nieuwsgierigheid naar cross-platform development samen met mijn passie voor Android leidde tot de keuze voor dit onderwerp. Dit onderzoek heeft mij veel bijgeleerd over de onderliggende werking van beide platformen alsook verschillende programmeerconcepten. Dit bracht mij een stap dichter bij de volwaardige programmeur die ik na deze opleiding wens te worden.

Graag zou ik van dit voorwoord ook nog gebruik willen maken om een aantal mensen te bedanken. Eerst en vooral zou ik Navaron Bracke willen bedanken. Zijn kennis, hulp en feedback werden enorm geapprecieerd en hielpen mij een andere kijk werpen op het onderzoek. Vervolgens zou ik mevrouw Akin willen bedanken voor de tijd en moeite die zij stak in het lezen en verbeteren van deze bachelorproef. Ook wil ik Next Apps bedanken voor het voorleggen van het onderwerp. De kennis die zij mij bijbrachten over Android was van grote hulp tijdens dit onderzoek. Daarnaast wil ik Kilian Hoefman persoonlijk bedanken voor de hulp bij dit onderzoek maar ook voor de steun doorheen de jaren. Tot slot wil ik mijn vrienden en familie bedanken voor alle hulp en steun doorheen deze opleiding. Deze mensen hielpen deze Bachelorproef vormen tot wat hij nu is.

Samenvatting

Vier jaar geleden werd Flutter uitgebracht door Google. In deze vier jaar werd Flutter de meest gebruikte cross-platform ontwikkelings-stack. Het overtreft zijn concurrentie, sommige die al meer als 10 jaar bestaan.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

De impact van mobiele applicaties (apps) op ons alledaags leven is niet te onderschatten. Ze zijn de dag van vandaag niet meer weg te denken. Dit komt door de grote toename in de verkoop van smartphones. Ongeveer 40% van de wereldbevolking beschikt over een smartphone en in 2020 gingen 1.38 miljard nieuwe toestellen de deuren uit. De toename in verkochte smartphones leidt tot een grotere markt voor apps. Dit hebben softwarebedrijven niet over het hoofd gezien. Het aantal applicaties aanwezig op de iOS App Store en de Google Play Store, is in de voorbije tien jaar respectievelijk vertwintig- en verdertigvoudigd. Elke smartphone maakt gebruik van een besturingssysteem, dit is de software die de hardware aanstuurt. Tegenwoordig is er een oligopolie van twee spelers op de markt van smartphone besturingssystemen (vanaf nu OS voor Operating System). De eerste speler, Apple ontwikkelde hun eigen OS genaamd iOS terwijl de meeste andere merken zoals Samsung en OnePlus gebruik maken van het Android OS. Elk OS heeft een verschillende manier van applicatieontwikkeling. In de meeste gevallen zullen app eigenaars met hun applicatie een zo breed mogelijk publiek willen bereiken. Hierdoor zullen ze apps moeten ontwikkelen voor zowel iOS en Android.

Wat is Native Development?
Als het ontwikkelingsteam kiest om een applicatie *native* te gaan ontwikkelen dan wil dit zeggen dat een aparte *codebase* (een verzameling broncode die gebruikt wordt om een bepaald softwaresysteem, applicatie of softwarecomponent te bouwen) moet ontwikkeld worden voor elk mobiel besturingssysteem waar de app moet op kunnen draaien. Verschillende codebases wil zeggen meer ontwikkelings- en onderhoudswerk maar geeft anderzijds wel een optie tot een native- design en gedrag per codebase.

Wat is Cross-platform Development?
Een andere aanpak is het ontwikkelen van een cross-platform app. Dit is het schrijven van één codebase die kan gecompileerd worden naar verschillende besturingssystemen. Het is een optie die de laatste jaren meer en meer gekozen wordt. De voor- en nadelen die hiervoor zijn aangehaald, draaien zich om. Een enkele codebase is goedkoper om te bouwen en te onderhouden maar dit moet dan ook afgewogen worden tegen het laten van native- designs en gedrag. Er zijn verschillende ontwikkeltools gemaakt voor het maken van cross-platform apps. Een paar van de grootste zijn React Native, Xamarin, Flutter en Ionic.

Wat is Flutter
Flutter is een Google UI toolkit voor het bouwen van mooie, bijna natively gecompileerde apps voor mobiel, web, en desktop en dit alles van één enkele codebase. Flutter kondigde op 3 maart 2021 hun nieuwe update aan die hen weer een stap in de goede richting duwde. Dit is onder andere te danken aan de grote steun en de snelle tractie van het platform in de voorbije jaren. Flutter is opensource wat wil zeggen dat ze hun broncode openstellen voor het publiek, waardoor iedereen het kan bekijken. Zo kunnen verbeteringen worden gedaan door eindgebruikers die het beste voor hebben met het platform. Als Flutter deze verbeteringen goedkeurt worden deze opgenomen in de broncode en beschikbaar gesteld voor iedereen.

Waarom Flutter versus native Android?
Flutter en Android zijn beiden ontwikkeld door Google. Alhoewel Flutter relatief jong is, kan dit niet gezegd worden van native Android. Het platform kende al veel updates en staat al lang niet meer in zijn jonge schoentjes. Vandaar dat het toetsen van deze twee een interessante kijk kan geven op het jonge flutter platform. De Flutter applicatie zal geschreven worden voor- en alleen gecompileerd worden naar Android. De sterkte van Cross-platform development is natuurlijk één codebase voor verschillende besturingssystemen, maar voor dit onderzoek is alleen Android van belang. Het onderzoek zal rekening houden met het geschreven aantal lijnen code per uitgewerkt hoofdstuk. Als het onderzoek rekening moet houden met iOS brengt dat het evenwicht uit balans.

Waarom Kotlin Android en geen Java Android?
Deze keuze is voor velen persoonlijk alhoewel hier ook onderzoek naar verricht is. In deze studie wordt Kotlin gebruikt omdat de syntax van de taal zal helpen bij het intomen van het aantal lijnen code. Hoe minder lijnen code, hoe eenvoudiger en overzichtelijker de app. Het is voor velen dan ook de voorkeurstaal om Android applicaties in te ontwikkelen. Dit onderzoek zal geen stap voor stap uitleg inhouden van hoe de code in elkaar zit. Het doel is een vergelijking schetsen tussen Flutter en Android. De code die wordt geschreven

zal dus alleen worden aangehaald indien dit relevant is voor de studie.

1.1 Probleemstelling

Het te onderzoeken domein kwam vanuit Next Apps. Een native app ontwikkelingsbedrijf uit Lokeren dat zich specialiseert in Android en iOS watch, phone en tablet apps. Zij merken dat niet alle klanten op de hoogte zijn van het hoge kostenplaatje van een native ontwikkelde app. Om in de toekomst een middenweg te vinden en de kost van een kleine app te dempen, denken zij erover om sommige apps cross-platform te gaan ontwikkelen. Bij het onderzoeken van de cross-platform markt kwamen ze Flutter tegen, het nieuwe platform met veel potentieel. Daarom werd de vraag gesteld om de stand van het Flutter platform te onderzoeken. Met deze scriptie hopen ze een duidelijk beeld te krijgen op de limieten en mogelijkheden van het framework.

1.2 Onderzoeksvraag

1.2.1 Hoofdonderzoeksvraag

Wat zijn de voor- en nadelen van app ontwikkeling in Flutter in vergelijking met native Android?
Het onderzoek is gericht op het vinden van een antwoord op de hoofdvraag. Het antwoord op deze vraag schetst een duidelijk beeld van de huidige stand van Flutter in vergelijking met Android. Hierbij wordt gelet op de ontwikkelingstijd van de features, de complexiteit en toegankelijkheid van de code, het aantal lijnen geschreven lijnen code. Hypothese: de voorkeur van ontwikkelingsplatform zal hoogstwaarschijnlijk afhangen van het soort app dat ontwikkeld moet worden. Een grotere app met veel features zal eerder native ontwikkeld worden, terwijl kleinere apps die weinig native gedrag bevatten waarschijnlijk cross-platform ontwikkeld zullen worden. De voor- en nadelen van beide platformen zullen talkrijk zijn, echter zullen de voordelen van Flutter groter zijn dan die van Android.

1.2.2 Deelonderzoeksvragen

Is Flutter al matuur genoeg om te aanschouwen als volwaardig alternatief op native app ontwikkeling?
Deze vraag biedt een antwoord aan alle native app ontwikkelaars die denken om de overstap te maken naar cross-platform development. Het is belangrijk om een overzicht te krijgen in de limieten van een platform alvorens tijd en geld te investeren. Hypothese: Flutter zal geen duidelijke standaard hebben. Best practices, goed ondersteunde libraries en coding principles zijn nog niet gedefinieerd door het platform waardoor het voor beginners onduidelijk zal lijken wat de beste manier van aanpak zal zijn. Ondanks de snelle groei en de grote steun van het platform wordt wel verwacht dat hier snel verandering in gebracht wordt.

Is Flutter toegankelijk voor nieuwe ontwikkelaars?
Het proces van het ontwikkelen van de Flutter applicatie zal antwoord bieden op deze vraag. Voor velen is het moeilijk om te schakelen naar een nieuw platform met een nieuwe taal. Om deze overgang transparant te maken, wordt een beeld geschetst van de moeilijkheidsgraad van Flutter en Dart. Hierdoor kan de lezer zelf beslissen of hij deze keuze wil maken. Hypothese: voor ontwikkelaars met een Java of C++ achtergrond zou de Dart taal geen probleem mogen vormen. Ze hebben een gelijkaardige syntax. De verwachtingen van het Flutter platform liggen hoog. De snelle groei van de gebruikersbasis doet vermoeden dat het een toegankelijk platform is.

1.3 Onderzoeksdoelstelling

Zoals hiervoor vermeld zal in deze scriptie onderzoek worden gedaan naar de voor- en nadelen van het Flutter framework. Om dit op een duidelijke manier te doen, zal het framework getoetst worden aan native Android. Door het jonge platform af te wegen tegen het ontwikkelde native tegengestelde, wordt gehoopt op een duidelijk beeld van Flutter. Het zal interessant zijn om te weten of het platform te kort doet aan native Android. Het is interessant om weten of de gevestigde waarde Android zijn troeven kan uitspelen tegenover de nieuwkomers. Het onderzoek wordt gevoerd aan de hand van een vergelijkende studie.

Voor het onderzoek werden twee paar identieke applicaties ontwikkeld. Al deze applicaties werden ontwikkeld in Android Studio. Beide paren bevatten één applicatie geschreven in de Kotlin taal voor native Android en één in Dart taal voor Flutter. Door het ontwikkelproces van de Flutter apps af te toetsen tegen dat van de native Android apps zal een beeld worden geschetst van de stand van het Flutter framework. Met dit beeld zal dan een antwoord gevormd worden op de onderzoeksvraag.

Dit onderzoek gaat gepaard met een deadline, deze tijdsdruk zorgt ervoor dat een aantal criteria moeten gesteld worden alvorens te starten. Deze criteria zorgen voor een kwaliteitsvol eindresultaat in een haalbaar tijdsbestek. Het eerste aantal criteria gaat over de performantie van de twee apps. Dit criterium wordt verder onderverdeeld in drie verschillende criteria. Allereerst wordt gekeken naar de opstartsnelheid van de apps. Vervolgens wordt de grootte van de uitvoeringsbestanden bekeken. Als laatste wordt gekeken naar het CPU gebruik van beide apps. De volgende worden hieronder kort aangehaald.

Het beoogde resultaat van de studie biedt een antwoord op alle onderzoeksvragen. Indien dit lukt, kan de studie als succesvol worden beschouwd, anders zal een suggestie worden gedaan voor toekomstig bijkomend onderzoek.

1.4 Opzet van deze bachelorproef

Deze scriptie kan opgedeeld worden in verschillende hoofdstukken, elk hoofdstuk kan bestaan uit verschillende secties of ondertitels. Deze sectie beschrijft de inhoud van elk van deze hoofdstukken.

Hoofdstuk 1: Inleiding: schetst een kort beeld van de inhoud van deze studie. Biedt een antwoord op sommige vragen rond deze paper.

Hoofdstuk 2 Stand van zaken: bevat een literatuurstudie. Een samenvatting van relevante onderzoeken die reeds voorafgingen aan dit onderzoek.

Hoofdstuk 3 Methodologie: omschrijft hoe het onderzoek in zijn werk zal gaan. De opzetting van het onderzoek en het onderzoek zelf zullen duidelijk toegelicht worden.

Hoofdstuk 4 Performantie: schetst een vergelijking tussen de platformen op basis van performantie criteria. Dit hoofdstuk is verder onderverdeeld in sub-criteria.

Hoofdstuk 5 Asynchroon werken: vergelijkt de meest gebruikte manieren van asynchroon werken in beide platformen.

Hoofdstuk 6 Gebruik van online APIs: bekijkt de werking van http verzoeken voor beide platformen.

Hoofdstuk 7 App veiligheid: omvat een lijst van best-practices met daarbij kort uitgelegd hoe ze geïmplementeerd dienen te worden in beide platformen.

Hoofdstuk 8 Code complexiteit: kijkt naar de code complexiteit van beide platformen aan de hand van de geschreven applicaties.

Hoofdstuk 9 Tooling: omvat een lijst van meest gebruikte command line tools voor elk platform en vergelijkt de tooling opties.

Hoofdstuk 10Appendix: zal verdere uitleg bieden rond bepaalde onderwerpen voor de geïnteresseerde lezer.

Hoofdstuk 11 Conclusie: dit is de algemene conclusie op deze scriptie. In deze conclusie zal een antwoordt gegeven worden op de reeds aangehaalde onderzoeksvragen. Daarbij wordt ook aangezet tot toekomstig onderzoek binnen dit vakgebied.

2. Stand van zaken

Dit hoofdstuk is equivalent aan een literatuurstudie. De eerste stap in dit onderzoek was het zoeken naar voorgaande studies over Flutter en Android. Elke studie die voldeed aan de eisen en als interessant werd beschouwd, werd opgeslagen en gebundeld. Hiervan werd de inhoud diagonaal gelezen om zo alleen de meest relevante onderzoeken over te houden.

Vervolgens werd een opsomming gemaakt en een kort beeld geschetst van een aantal van deze studies en hun bijdrage in het vakgebied. In de volgende sectie zullen dus alleen de meest relevante studies, die dicht aansluiten bij dit onderzoek, worden aangehaald. De ondervindingen uit deze studies werden gebruikt als voorbereiding op het onderzoek en kunnen in volgende hoofdstukken gebruikt worden als steunpunten. Door de ondervindingen van verschillende studies te bundelen, kon gezocht worden naar een rode draad.

De te onderzoeken criteria van deze studie werden vastgelegd op basis van de reeds voorgaande onderzoeken. Het is de bedoeling dat deze studie bijdraagt aan het vakgebied. Hierdoor wordt weerhouden van reeds onderzochte criteria opnieuw te gaan onderzoeken. Toch zal blijken dat sommige onderzochte criteria opnieuw worden bekeken. Deze herhaling komt voort uit de Flutter 2.0 release, deze zorgde voor veranderingen in de prestatiemogelijkheden van het framework.

2.1 Voorgaande onderzoeken

Navaron Bracke onderzocht de verschillen en gelijkenissen tussen Android en Flutter in een vergelijkende studie. (Bracke, 2020). De scriptie vormde een beeld over de toenmalige stand van het Flutter framework door het te toetsen tegen het ontwikkelde Android. Het onderzoek was afgebakend door een vooraf vastgelegd aantal criteria. Zo kon de onderzoeker zekerheid bieden over de grondigheid van de studie. De onderzochte criteria waren: Internationalisering, navigatie, persisteren van gegevens, user Interfaces, asynchroon werk, permissies, software testing, opstarttijd van de applicatie, grootte van de applicatie. De opzet en het doel van het onderzoek liep in grootte mate gelijk met het onderzoek gevoerd voor deze scriptie. Daarom was het onderzoek van Bracke uitermate interessant, bevindingen werden bijgehouden, genoteerd en getoetst tegen de bevindingen van dit onderzoek. De hoofdstukken die performantie onderzochten werden gebruikt om een beeld te schetsen van de Flutter evolutie over tijd. De andere twee hoofdstukken (User Interfaces en Asynchroon werk) werden opnieuw onderzocht met conclusies van Bracke in het achterhoofd. Hiervoor werd, in de mate van het mogelijke, zoveel mogelijk gefocust op de onderwerpen die Bracke niet aanhaalde. De onderzoeker constateerde dat, ondanks de jonge aard van het Flutter platform, het een goed alternatief biedt op elk onderzocht criterium. Bij elk van deze criteria werd een platform van voorkeur gekozen. Hieruit bleek Flutter op 4 van de 7 criteria naar voor te komen. Al werd wel meegedeeld dat niet alle features in Flutter even gebruiksvriendelijk zijn maar door de jonge aard werd verwacht dat dit in de toekomst zou verbeteren.

Daniel Dang en David Skelton, twee professors aan de Insitutue of Technology, schreven een wetenschappelijk artikel over mobile app development in het onderwijs (Dang & Skelton, 2019). Het doel van het onderzoek was het bepalen van het ideale framework voor het aanleren van mobiele applicatieontwikkeling in het onderwijs. Het onderzoek werd gevoerd aan de hand van verschillende native- en cross-platform frameworks. Zo werden native Android, native iOS, Flutter, React Native, Ionic, Xamarin, Cordova en Appcelerator vergeleken. Onderzochte criteria bij dit onderzoek waren: Design App User Interface, User Event Handling, Services & Multiple threads used in Internet connection, Graphics and Animation, Device hardware and sensor, Wireless connectivity, Internal data storage, Real time database. Dit artikel werd gebruikt als basis voor het sommige delen van het onderzoek gevoerd in deze scriptie. Zo werden de hoofdstukken User Event Handling en Services & Multiple threads used in Internet connection grondig gelezen en ontleed voor dit onderzoek. In conclusie bleken Flutter en Native Android de twee beste platformen te zijn voor gebruik tijdens praktijklessen. Ook beval het onderzoek acht ideale onderwerpen aan die in de praktijksessies moeten worden behandeld om studenten voldoende technische vaardigheden bij te brengen om alle soorten mobiele apps te ontwikkelen. Belangrijk hierbij te vermelden is dat de studie gebruik maakte van een oude Flutter release.

Austen Latture onderzocht de ontwikkelcyclus van een app in Flutter (Latture, 2020). De focus van het onderzoek lag hem op de leercurve van het platform. Hier waren geen vooropgestelde criteria aanwezig, deze studie omschreef de uitwerking van de Backdrop app. In conclusie bleek Flutter een toegankelijk platform te zijn voor nieuwe ontwikkelaars met een allesbehalve steile leercurve. Een ander groot voordeel aangeboden door

het platform is de talrijke ingebouwde functionaliteit. De Dart taal die wordt geschreven voor het maken van Flutter apps is volgens het onderzoek gemakkelijk aan te leren voor mensen met een achtergrond in de C taal. Aan de andere kant bleek het kiezen van third-party libraries een hinder blok te vormen tijdens de ontwikkelfase. De jonge aard van het platform ging gepaard met jonge libraries. Zoals bleek uit andere studies concludeerde dit onderzoek dat Flutter hier een rode draad in mist. De grootte hoeveelheid aangeboden libraries maakt het moeilijk om te weten welke goed ondersteund worden en dan ook vaak gekozen worden. Dit is vooral een struikelblok voor nieuwe developers, onbekend met het platform. Vervolgens werd Flutter ook in het kort vergeleken met React Native. Hieruit kon geconcludeerd worden dat Flutter veel gemakkelijker te installeren is. Dit komt, volgens het onderzoek, voort uit de ingebouwde functionaliteit van Flutter terwijl React meer werkt met verspreide derde partij software die meestal bestaat uit te veel boilerplate code. Het onderzoek van Latture hielp dit onderzoek met antwoorden op één van de deelonderzoeksvragen 'Is Flutter toegankelijk voor nieuwe ontwikkelaars?'. Ondanks de subjectiviteit van de paper, werd wel rekening gehouden met de bevindingen van het onderzoek. Echter kon niet naast de onwetenschappelijke aard van het onderzoek gekeken worden, hierdoor wordt verder niet gerefereerd naar deze bevindingen.

Carlos Chavez, een student en Yoonsik Cheon, een professor aan de Universiteit van Texas El Paso onderzochten het herschrijven van Native Android apps naar Flutter apps (Cheon & Chavez, 2020). Het onderzoek bekeek een native Android app ontwikkeld in de Java taal en probeerde deze zo goed mogelijk om te zetten naar een Flutter app die terug kon gecompileerd worden als een Android app maar ook als iOS app. Het onderzoek werd gevoerd aan de hand van het Flutter ontwikkelproces. De technische-, praktische problemen en uitdagingen die de kop opstaken werden vervolgens besproken. Het onderzoek deed een paar duidelijke verschillen uit blijken tussen Android en Flutter wanneer gekeken werd naar de taal, programmeerstijl en design. Het grootste werk was het herschrijven van de views terwijl het herschrijven van de achterliggende logica eerder snel gebeurd was. Het onderzoek focuste zich onder andere op het aantal lijnen code geschreven voor beide applicaties. De Flutter code bleek hierbij maar 2/3 van het totaal aantal Android code te zijn. Maar volgens de onderzoekers was dit niet genoeg om apps in Flutter te beginnen ontwikkelen. De leeftijd van het platform bracht nog te veel onzekerheid met zich mee. Ook ontbrak dit onderzoek aan een soort standaard werkwijze of verzameling van richtlijnen. Dit sloeg dan vooral op libraries en APIs binnen het platform. De gebruikersbasis en de beschikbare tools moesten hierbij ook zeker in rekening worden gebracht. Het grote aantal third party libraries, door de grootte aanhang, zorgde voor onzekerheid bij het zoeken naar een goed ondersteunde library. Bij andere grootte platformen wordt dit verholpen doordat na bepaalde duur de beste APIs en libraries naar boven komen en een soort van community standard wordt gezet. Dit onderzoek was uitermate interessant en legde gronde voor een groot deel van dit onderzoek. Ook al zijn geen raakvlakken aanwezig tussen de twee onderzoeken, veel kennis vergaard uit deze paper werd gebruikt als bouwsteen van dit onderzoek.

Mathilda Olsson onderzocht hoe Flutter applicaties vergelijken met native applicaties (Olsson, 2020). Opnieuw een onderzoek dat dicht aansluit bij het onderzoek gevoerd voor deze scriptie. Voor haar onderzoek ontwikkelde Olsson twee native applicaties, één in Kotlin Android en één in Swift iOS. Deze applicaties werden vervolgens beoordeelt op

vlak van CPU performantie. Uit onderzoek bleek dat Flutter en Android apps gelijkaardige resultaten halen op vlak van CPU performantie. Vervolgens werd gekeken naar het aantal lijnen code en de complexiteit van de twee code bases. Hieruit bleek dat Flutter aanzienlijk minder lijnen code nodig had maar ook minder complexe code bevatte. De Flutter app bestond uit 125 lijnen code terwijl de twee native apps samen 580 lijnen code bevatte. Dit verschil is substantieel en duidt erop dat Flutter beter blijkt voor kleine tot middelgrote applicaties. De bevindingen over CPU performantie werden gebruikt in dit onderzoek voor het schetsen van een beeld van CPU performantie over tijd. Daarbuiten bleek Flutter opnieuw een betere optie over Android, wat leidde tot het vormen van een hypothese op de onderzoeksvraag. Het onderzoek van Olsson bevatte een bevraging die zich focuste op de verschillen in gebruikersperceptie. Een aantal gebruikers moesten de app eerst testen waarna ze een vragenlijst konden beantwoorden waaruit verschillen in look en feel moesten duidelijk worden. Uit de vragenlijst bleek native ontwikkeling, op vlak van UI, de voorkeur te hebben van het grootste deel gebruikers. De verschillen waren vooral te zien bij animaties, fonts, gedrag, lijsten en menus. Animaties konden wel gelijkgesteld worden maar dat is extra werk zijn voor de ontwikkelaar. In conclusie, zoals in de andere onderzoeken, werd nogmaals geduid op de leeftijd van het platform. Alle criteria waar Flutter minder op scoorde konden nog volop in ontwikkeling zitten voor een volgende release. Flutter had volgens het onderzoek veel potentieel indien de steun van de gebruikersbasis groot bleef.

Jakhongir Fayzullaev onderzocht drie verschillende cross-platform development frameworks, Kotlin Native, Multi-OS en Flutter in een vergelijkende studie (Fayzullaev, 2018). De frameworks en dus de studie is vooral interessant voor Android ontwikkelaars aangezien de onderzochte frameworks dicht aansluiten bij de Java en Android ontwikkeling methoden. In 2018, toen het onderzoek werd uitgevoerd, waren de Flutter en Multi-OS nieuwe spelers op de cross-platform markt. Het onderzoek kon zich niet baseren op voorgaande studies en ontbrak aan duidelijke richtlijnen van de frameworks. Hierdoor was het onderzoek oppervlakkig, het vergeleek de grote lijnen van de platformen zonder de details te bekijken. Het Flutter deel van het onderzoek bekeek in grote lijnen: Widgets, het maken van layouts en de bouwstenen van Flutter. Fayzullaev concludeerde dat op vlak van performantie de drie platformen zo goed als identiek waren. Onderzoek deed blijken dat Flutter een goed platform was in 2018 maar de jonge aard zorgde voor weinig bruikbare libraries en tools. Ook had Flutter weinig built-in functionaliteit met als gevolg meer werk voor de programmeurs die meer code moesten schrijven ten opzichte van de andere meer ontwikkelde platformen. Fayzullaev omschrijft deze manier van coderen als het opnieuw uitvinden van het wiel. Ondanks het ontbreken van detail in dit onderzoek werd het wel gebruikt om kennis op te doen over de bouwstenen van Flutter.

Sebastian Faust schreef een Bachelorproef over het maken van grootschalige Flutter applicaties (Faust, 2020). Het doel van dit onderzoek is andere applicatieontwikkelaars helpen wanneer zij voor dezelfde problemen komen te staan tijdens het schrijven van grote Flutter apps. Veel van de andere onderzoeken concludeerden dat Flutter beter is voor kleinschalige applicaties. Daarom was dit onderzoek interessant om op te nemen in deze literatuurstudie. Het vormde een benchmark onderzoek voor de schaalbaarheid van Flutter applicaties. Voor dit onderzoek schreef Faust een app met veel complexe features, tijdens dit proces documenteerde hij alle design keuzes en de problemen/obstakels die

hieruit voortkwamen. Na het evalueren van elk obstakel kon hij altijd tot een optimale oplossing komen. Na de studie werden ondervindingen gebundeld en een set van richtlijnen opgesteld voor het ontwikkelen van grote Flutter applicaties. Richtlijnen en best practices waar developers zich best aan houden tijdens het ontwikkelingsproces. Deze werden gebundeld in een gids die later werd gepubliceerd en goed ontvangen werd door de Flutter community. Voor verdere ondersteuning van het onderzoek werd nog een interview afgenomen met een Flutter expert. De conclusie van de paper toonde de goede schaalbaarheid van Flutter apps aan maar dit onderzoek werd vooral gebruikt voor het oplossen van hinderblokken die Faust gedetailleerd onderzocht

Als laatste is het interessant om kort de nieuwe Flutter update aan te halen (Flutter, 2021). De impact hiervan op dit onderzoek en de verschillen met vorige studies te duiden. Op woensdag 3 maart 2021 werd Flutter Engage georganiseerd, een live event gebracht door het Flutter team. Niet alleen het Flutter platform kreeg een upgrade maar ook de taal Dart werd verbeterd voor een vlottere en vertrouwelijke programmeer ervaring. De grootste Flutter upgrade was de overgang van een mobiel framework naar een portable framework. Oorspronkelijk werd Flutter alleen gecompileerd voor iOS en Android besturingssystemen maar door de upgrade zijn daar: Windows, macOS, Linux en Web bijgekomen. Echter valt dit buiten de scope van dit onderzoek. De grootste Dart upgrades worden hieronder opgelijst:

- Sound null safety is toegevoegd, zonder oude code te breken. Dart zal zeggen waar mogelijks gevaren zitten op null excepties.
- Smarter flow analysis
- Late variables
- Required named parameters
- DevTools upgrade

Ook werden in de keynote verschillende grote bedrijven zoals Ubuntu, Microsoft, Toyota genoemd die in de toekomst nauw gaan samenwerken met Flutter. Deze samenwerkingen tonen de kracht, potentieel en groei van het platform. Ook werd in de Keynote het aantal open issues op Github aangehaald. Doordat het een opensource framework is kan iedereen die de broncode wil zien, deze gaan opzoeken op GitHub. Moest een fout stuk code, een simpeler te schrijven stuk code of zelfs als een goede bijdrage aan het platform gevonden worden kan hier een issue van gemaakt worden. Als Flutter deze verbetering heeft nagekeken en goedgekeurd wordt deze toegevoegd aan de broncode. Op deze manier sparen zichzelf werk uit en groeit het platform snel en volgens de wensen van de gebruikers. Het grootte aantal issues die momenteel open staan zien zij dan ook als een teken van groei. De vele gebruikers die Flutter willen beter maken is een teken dat er veel vertrouwen is in het platform. Door de updates en aanpassingen zullen veel APIs verouderd zijn. Flutter heeft dit echter goed geanticipeerd door het flutter fix commando toe te voegen. Deze zal de bestaande code doorlopen en tonen welke APIs verouderd zijn en in wat ze best vervangen worden.

2.2 Waarom deze studie?

Cross-platform development heeft een snelgroeiende gebruikersbasis. De vele updates en verbeteringen aan de platformen maakt cross-platform development dan ook elke dag aantrekkelijker. Dit leidt tot meer onderzoek en studies binnen de sector. Tijdens de literatuurstudie bleek dat een groot aantal papers Flutter vergeleek met andere cross-platform ontwikkel tools. Dit soort papers moet de lezer helpen bij het maken van een keuze tussen deze tools. Door het verdelen van de aandacht over verschillende platformen, komt Flutter niet uitgebreid aanbod in dit soort papers. Voor 2020 waren de meeste onderzoeken van deze aard. Flutter leek toen nog niet stabiel en groot genoeg om als afzonderlijke tool onderzocht te worden.

Zoals uit voorgaande sectie blijkt, is Flutter meerdere malen onderzocht in 2020. In verschillende studies werd het tekort aan onderzoek voor 2020 aangehaald. Flutter is in het afgelopen jaar in een stroomversnelling geraakt op vlak van onderzoek. Dit is te danken aan de sterke stijging in gebruikersaantal. Het platform is nog steeds relatief jong maar de aantrekkingskracht ligt hem in de gebruikersbasis en de grote steun. Hoe meer gebruikers, hoe belangrijker het product in kwestie. Hoe belangrijker het product, hoe meer middelen hierin worden geïnvesteerd en dat lokt op zijn beurt weer meer gebruikers. Deze cirkel wordt alleen onderbroken als het platform in kwestie zijn aantrekkelijkheid verliest. Flutter lijkt hier geen last van te hebben en deze studie wil laten blijken waarom.

Dus kan geconcludeerd worden dat Flutter een interessant onderzoeksdomein is. Maar zoals bij elk onderzoek binnen de IT, zijn bevindingen al snel verouderd. De rappe evolutie binnen de sector doet elk product streven naar de beste versie van zichzelf. De competitie is hoog, dus moet performantie altijd voorop staan. Bij elke update of verbetering zijn voorgaande performantie studies verouderd en niet accuraat. Hierdoor zijn de studies, aangehaald in dit hoofdstuk, dan ook vaak zo recent mogelijk. Toch onderging Flutter net een grote release. Vele zaken werden verbeterd, onder andere de performantie van het platform. De voorgaande studies zijn weer verouderd en hier ligt een kans om opnieuw onderzoek te doen en beeld te schetsen van de huidige performantie van Flutter. Ook lijkt het interessant om een beeld te schetsen van de performantie van Flutter over tijd.

3. Methodologie

Om een antwoord te kunnen bieden op zowel de primaire, als de deelonderzoeksvragen, werd een onderzoek verricht. Dit onderzoek bestaat uit twee grote delen. Het eerste deel is de literatuurstudie die in vorig hoofdstuk beschreven werd. Het tweede deel is een experiment waaruit ondervindingen werden geanalyseerd en samengevat in volgende hoofdstukken. Om inzicht te krijgen in de opzet van het experiment wordt hieronder de opbouw hiervan uitgelegd. Ook wordt elk onderzoekscriterium toegelicht.

3.1 Opzet Experiment

Zoals reeds vermeld, valt dit experiment onder het luik vergelijkende studie. Het gaat twee ontwikkeltechnieken met elkaar vergelijken. Om deze technieken op een correcte manier te vergelijken werden verschillende applicaties ontwikkeld.

Applicaties
Voor het experiment werden vier applicaties ontwikkeld. Twee applicaties met de Flutter stack en twee met de Android stack. Het eerste paar applicaties dat vergeleken werd waren de Hello world applicaties. Deze werden gebruikt om twee van de drie performantie criteria te onderzoeken. Voor het criterium CPU gebruik en de andere criteria werd een tweede paar applicaties ontwikkeld, verder wordt hiernaar gerefereerd als onderzoeksapplicatie. Het ontwikkelproces van deze applicaties werd aanschouwt als onderzoek. De vergelijking van de resultaten vormde het antwoord op de onderzoeksvraag.

Omgeving
De applicaties werden ontwikkeld in de Android Studio Integrated Development Enviroment (IDE). Android Studio biedt snelle tools voor het bouwen van apps op elk type Android-apparaat. Daarnaast werd tevens voor Android Studio gekozen gezien de uitgebreide mogelijkheden, het beperkt houden van ontwikkeltools alsook het gegeven dat dit platform gratis te gebruiken is. Andere mogelijke IDE's waren, voor Android IntelliJ IDEA en voor Flutter Visual Studio Code. Een vergelijking tussen deze IDEs valt echter buiten de scope van dit onderzoek.

Talen
De Android applicatie werd geschreven in de Kotlin taal. Kotlin is een gratis, open source programmeertaal ontworpen door JetBrains voor Java Virtual Machine en Android. De Flutter applicatie werd ontwikkeld in de Dart taal. Dit is een onafhankelijke taal ontwikkeld door Google maar ze wordt vooral gebruikt voor de ontwikkeling van Flutter applicaties.

Lijnen code
Tijdens de uitwerking van elk onderzoekscriteria werd rekening gehouden met een aantal sub-criteria. Op basis van deze sub-criteria werd een conclusie gevormd over de toegankelijkheid van beide frameworks. Eerst werd gekeken naar het aantal geschreven lijnen code. Het aantal lijnen code staat niet garant voor een betere sensatie van de app. Doch is het interessant om te kijken welke zaken bij het ene platform al dan niet uitgebreider dienen geïmplementeerd te worden om hetzelfde resultaat te bekomen.

Libraries of packages
Libraries in Android en packages in Flutter zijn een belangrijk en extreem krachtig aspect van ontwikkeling. Een library of package (verder library), is een verzameling van code die bepaalde functionaliteit reeds uitgewerkt heeft en dus zorgt voor snelle herbruikbare code. Omdat dit de snelheid van ontwikkeling kan beïnvloeden en tevens de robuustheid van de applicatie kan verbeteren, is het belangrijk om hier ook aandacht aan te besteden. Libraries kunnen de robuustheid van een applicatie verbeteren gezien de meeste libraries vaak open source zijn. Op deze manier worden nieuwe functionaliteiten toegevoegd die nodig blijken in verschillende use cases. Het gebruik van libraries kan drastisch helpen met de code complexiteit en het aantal geschreven lijnen code. Libraries nemen vaak een deel van de complexe code op zich. Echter is het niet altijd gemakkelijk om robuuste en goed ondersteunde libraries te vinden.

3.2 Onderzoekscriteria

In volgende hoofdstukken worden de onderzoekscriteria één voor één onderzocht. De alinea hieronder legt uit hoe deze criteria werden opgebouwd en vergeleken.

Onder het luik performantie vallen drie onderzoekscriteria. Deze worden ook wel de performantie criteria genoemd. Eerst hebben we de grootte van de uitvoeringsbestanden. Dit werd onderzocht aan de hand van de Hello World applicaties. Hierbij werden de

groottes van de apks van beide apps vergeleken. Vervolgens werd de opstartsnelheid van de twee Hello world apps onderzocht. Hierbij werden beide applicaties x aantal keer geopend en werd telkens de duur van de opstartprocedure genoteerd. Hiervan werd uiteindelijk het gemiddelde genomen en deze bevindingen voor zowel Android als Flutter werden dan met elkaar vergeleken. Daarna werd het CPU gebruik van beide applicaties onderzocht. Hierbij werd gebruik gemaakt van de onderzoeksapplicatie. Hierbij werden de percentages van CPU gebruik genoteerd wanneer door de app gelopen werd en deze uiteindelijk vergeleken.

Het volgende onderzochte criterium was het uit voeren van asynchrone taken binnen elk framework. Hierbij werden de verschillende manieren van asynchroon werken aangehaald voor elk framework. De voor- en nadelen van deze asynchrone ontwikkel methoden werden opgesomd, uitgelegd, onderzocht en vergeleken op basis van hun voor- en nadelen.

Het criterium daarna was gebruik van online API's. In dit hoofdstuk werd gekeken naar de mogelijkheden voor beide frameworks om API calls uit te voeren. Werd op een gelijke manier aangepakt als het hoofdstuk ervoor, asynchrone taken. Deze twee hangen samen aangezien netwerk verzoeken best asynchroon worden uitgevoerd aangezien deze lang duren.

Daarna werd het criterium app veiligheid onderzocht. Hier werden de best-practices rond app veiligheid opgelijst en onderzocht. Hier werden de verschillende manieren van implementatie van deze best-practices voor beide frameworks ook voorzien.

Daarna werd gekeken naar code complexiteit. Dit hoofdstuk maakte een vergelijking op basis van een algoritme en diende te verduidelijken welk platform en taal toegankelijker is voor nieuwe ontwikkelaars.

Als laatste werd tooling onderzocht. Dit hoofdstuk houdt een lijst van de belangrijkste CLI commando's in. Daaruit werd een vergelijking gemaakt tussen de frameworks en de mogelijkheden die ze bieden om taken zo wel/ zo niet uit te voeren via de command line.

3.3 Gebruikte hardware

Voor dit onderzoek werd gebruik gemaakt van bepaalde hardware. Sommige resultaten van het onderzoek, zoals performantie, zijn voor een groot deel afhankelijk van de onderliggende hardware. Om hier dus een duidelijke kijk op te krijgen werd de voor dit onderzoek gebruikte hardware hieronder opgesomd. Voor dit onderzoek werd geen gebruik gemaakt van een fysiek toestel maar van een emulator. Een emulator is hardware of software die een computersysteem in staat stelt zich te gedragen als een ander computersysteem.

Computer:
Toestel: Macbook Pro 2019
Processor: 2.6GHz 6-core Intel Code i7
Geheugen: 6 GB 2667 MHz DDR4

Emulator:
Toestel: Pixel 4
Android: 11.0, API level 30
CPU: x86_64

4. Performantie

Dit hoofdstuk zal het performantie aspect van de applicatie behandelen. Ondanks het feit dat nieuwe smartphones alsmaar performanter worden, is het nog steeds belangrijk om belang te hechten aan de performantie van een applicatie tijdens de ontwikkeling. Hetgeen evenredig stijgt met de performantie van nieuwe smartphones, zijn de gebruikersverwachtingen van de apps. Het is belangrijk dat app ontwikkelaars de applicatie zo schrijven dat deze de beste performantie biedt aan de gebruiker. Elke app ontwikkelaar heeft namelijk als hoofddoel het ontwikkelen van een app die veel gebruikt wordt en een goede ervaring bezorgd aan de gebruiker. Wanneer een app niet voldoet aan de gebruikerseisen zal deze stoppen met de app te gebruiken, deze verwijderen en misschien zelf een negatieve review achterlaten. Wat op zijn beurt leidt tot minder gebruikers en minder downloads.

Er zijn verschillende aspecten die de performantie van een app beïnvloeden. Deze kunnen onderverdeeld worden in verschillende categorieën. In dit onderzoek werd alleen gefocust op de app performantie. Hieronder vallen laadsnelheid, batterij verbruik, geheugen gebruik, grootte van het uitvoeringsbestand, geheugen gebruik van de app in achtergrond Dit onderzoek spitst zich toe op opstartsnelheid, grootte van de uitvoeringsbestanden en CPU gebruik. Ook moet rekening gehouden worden met de samenhang tussen performantie en gebruikte hardware. De gebruikte hardware voor dit onderzoek werd vermeld in vorige hoofdstuk.

Het is dus belangrijk om een performante app te ontwikkelen maar het gebruikte framework zit hier ook voor een deel tussen. Het doel van dit hoofdstuk is het vergelijken van deze onderliggende verschillen. Deze worden hieronder kort toegelicht.

4.1 De grootte van de uitvoeringsbestanden

Het eerste performantie aspect dat onderzocht werd was de grootte van het uitvoeringsbestand. Het doel is om een uitvoeringsbestand zo klein mogelijk te houden, zodat de gebruiker geheugen kan besparen op zijn toestel. Het is immers zo dat de prijs van high-end smartphones vandaag de dag vrij steil is. De consument kan bij het aanschaffen van een nieuw toestel vaak kiezen voor een bepaalde hoeveelheid opslag. Wanneer voor een toestel gekozen wordt met een vrij beperkte opslagcapaciteit en de gebruiker een aantal grote apps installeert, zal de opslagcapaciteit van het toestel snel volledig ingenomen zijn. Dit kan tot gevolg hebben dat de gebruiker bepaalde applicaties zal gaan verwijderen zodat hij terug ruimte kan creëren. Uit onderzoek bleek namelijk dat 1/2 van de applicaties verwijderd wordt wegens de grootte ervan. Dit is één van de redenen waarom het dus in de eerste plaats sowieso al een goed idee is om de app zo klein mogelijk te houden, maar de hoogste performantie te garanderen.

Een andere reden is de maximale toegestane grote van de uitvoeringsbestanden opgelegd door de verschillende app stores. Om een applicatie om de Google Play Store te kunnen beschikbaar stellen, moet deze voldoen aan meerdere eisen. Een van deze eisen is dat het uitvoeringsbestand van de app maximaal 100MB bedraagt. Het doel van een applicatie beschikbaar te stellen op de Play Store is immers om een zo groot mogelijk doelpubliek aan te spreken, wanneer echter niet aan de eisen van de Play Store voldaan wordt, zal dit doelpubliek niet bereikt worden. In dit hoofdstuk zal gekeken worden naar de grootte van de uitvoeringsbestanden van respectievelijk de Flutter applicatie alsook de Android applicatie.

4.1.1 Opzet

Een uitvoeringsbestand, ook wel gekend als een executable, van een Android app kan verschillende bestandsformaten zijn. De twee meest gebruikte formaten zijn Android Package (APK) en App Bundle. Deze bestanden worden gebruikt om een Android applicatie uit te voeren. Voor dit onderzoek werd gefocust op het APK bestandsformaat.

Ook werden voor dit onderzoek twee applicaties ontwikkeld en dus konden twee APKs gemaakt worden. De eerste in native Android en de tweede in Flutter. Het te vergelijken paar APKs waren van de Hello world applicaties. Om de verschillen tussen beide platformen te testen, werden de verschillende tegenhangers tegenover elkaar geplaatst. De Flutter APK werden met andere woorden vergeleken met hun native tegenhanger.

Voor de meest bruikbare resultaten werd gekozen om voor de applicaties een release build te maken. Elke build is een verzameling van een aantal regels die worden toegepast op de code wanneer deze compileert. De release build is een verzameling van regels die de app zo compact mogelijk maakt. Dit gebeurt door het opruimen en optimaliseren van code.

Android
In Android gebeurt dit aan de hand van volgende lijnen code in het app build.gradle bestand.

Tabel 4.1: Gebruikte maatstaf uitvoeringsbestanden

Bit	/	Binair getal, 1 of 0
Byte	B	8 Bit
Kilobyte	KB	10^3B
Megabyte	MB	10^6B

Listing 4.1: Android build.gradle (app)

```
release {
    minifyEnabled true
    shrinkResources true
    proguardFiles getDefaultProguardFile
    ('proguard-android-optimize.txt'), 'proguard-rules.pro'
}
```

Flutter
Voor een ideale Flutter release build wordt best gebruik gemaakt van onderstaande terminal commandos. Het flutter clean commando zal het project kleiner maken door het verwijderen van de build en .dart-tool mappen. Het commando daarna maakt een release APK per ABI. De APKs worden gesplitst per ABI omdat dit het uitvoeringsbestand aanzienlijk verkleint.
flutter clean
flutter build apk –split-per-abi
De resultaten van het onderzoek zijn bekomen gebruik makend van de apkanalyzer, een ingebouwde Android Studio tool.

Als laatste werd een opsomming gemaakt van de grootte van Flutter uitvoeringsbestanden over tijd. Dit moet een inzicht bieden in de evolutie van Flutter.

4.1.2 Resultaten

Een APK is een map of verzameling van bestanden. Al deze bestanden samen zorgen ervoor dat een applicatie kan worden uitgevoerd. Tabel 4.1 biedt inzicht in de gebruikte maatstaf.

Android Een Android app heeft geen minimum grootte voor het uitvoeringsbestand. Echter als de app op de Play Store dient gezet te worden moet deze minimum 7KB bedragen. Wanneer een nieuw Native Kotlin Android project wordt opgezet met één activity, zal dit automatisch een Hello World applicatie zijn. Als eerste stap in het onderzoek werd gekeken naar de omvang van deze APK. Volgens apkanalyzer was deze APK 3.2 MB in omvang en 2.6 MB voor de download. Vervolgens werd gekeken naar een verkleinde versie van de codebase waaruit de testing directories verwijderd werden samen met de bijhorende dependencies. Vervolgens werd minifyEnabled als ook shrinkResources op true gezet. Dit leidt tot een APK van 1.6 MB met een downloadgrootte van 1010.1KB. Voor een derde en finale versie van de APK werd de codebase nog extra verkleint. Dit be-

Tabel 4.2: Grootte van de uitvoeringsbestanden in Android

	APK size	**Download Size**
META-INF	3 KB	3 KB
res	123.7 Kb	117 KB
AndroidManifest.xml	721 B	721 B
classes.dex	293.9 KB	293.1 KB
Resources.arsc	222.6 KB	51.7 KB
Kotlin	9.1 KB	9 KB
Total	**693.2 KB**	**476.1 KB**

Tabel 4.3: Grootte van de uitvoeringsbestanden in Flutter

	ARM		**ARM-64**	
	APK Size	**Download size**	**Apk Size**	**Download Size**
lib	4.2 MB	4.2 MB	4.7 MB	4.6 MB
assets	183.4 KB	183 KB	183.4 KB	183 KB
META-INF	7.9 KB	7.6 KB	7.9 KB	7.6 KB
res	6.2 KB	5.9 KB	6.2 KB	5.9 KB
AndroidManifest.xml	1021 B	1021 B	1022 B	1022 B
classes.dex	121.7 KB	121.3 KB	121.7 KB	121.3 KB
Resources.arsc	22.6 KB	3.8 KB	22.6 KB	3.8 KB
Kotlin	9.7 KB	9.7 KB	9.7 KB	9.7 KB
Total	**4.6 MB**	**4.5 MB**	**5 MB**	**5 MB**

tekende ook dat niet meer werd gewerkt volgens Android best practices. Tabel 4.2 bevat de resultaten van dit onderzoek. De code voor deze template kan terug gevonden worden op GitHub repository.

Flutter

Flutter maakt gebruik van de Flutter Engine. (zie appendix 10.2) Deze engine maakt deel uit van de APK en is nodig voor het gebruiken van een Flutter app. Deze engine bevat het gehele framework en is daarom een aantal megabyte groot. Dit maakt de Flutter APK al direct meerdere megabytes in omvang. Voor het eerste deel van het onderzoek werd gekeken naar een standaard versie van de Hello world template. Hierbij bevatte de main.dart file 26 lijnen code. Hierbij was de APK niet gesplitst op abi 15.5 MB groot terwijl de gesplitste APK 5.1 MB bedroeg voor de gewone ARM en 5.5 MB voor de ARM64. Vervolgens werd de applicatie herschreven met als doel een zo klein mogelijke APK. Hierbij was het main.dart bestand 12 lijnen code na het uitvoeren van een code format. De APK versie waarbij niet gesplitst werd per abi was 14 MB groot. De resultaten van de APK gesplitst op abi staan vermeld in tabel 4.3. De code voor dit onderzoek kan terug gevonden worden op de GitHub repository.

Tabel 4.4: Grootte van de uitvoeringsbestanden in Flutter over tijd

Tijd	Download Size
Maart 2018	4.06 MB
Augustus 2018	4.20 MB
Begin Oktober 2018	4.28 MB
Eind Oktober 2018	4.48 MB
November 2018	4.70 MB
December 2018	6.70 MB

Op de Flutter site staat een korte omschrijving van de downloadgrootte gemeten van een minimale Flutter-app (geen materiële componenten, slechts een enkele Center-widget, gebouwd met flutter build apk –split-per-abi), gebundeld en gecomprimeerd als een release-APK. Deze minimale downloadgrootte werd in 2018 verschillende malen verkleint en opnieuw gemeten. Tabel 4.4 bundelt de resultaten van deze metingen.

4.2 De opstartsnelheid van de applicatie

Een andere veel voorkomende reden waarom gebruikers applicaties slecht beoordelen of verwijderen is een trage applicatie. Onderzoek toont aan dat gebruikers verwachten dat een applicatie binnen maximaal drie seconden opstart. Met opstartsnelheid wordt de tijd bedoelt tussen het drukken op het app icoon en het te zien krijgen van een volledige view. Hierbij moet rekening gehouden worden met de in te laden data. Het inladen van data heeft in veel gevallen weinig te maken met app performantie. In veel gevallen zal dit te maken hebben met de aangesproken API. Een onderzoek naar de respons tijd van een API valt echter buiten het bestek van dit onderzoek.

Het verschil tussen het inladen van data en het opstarten van de app kan gezien worden aan de hand van een laad icoon. Als de app een API aanspreekt voor data zal een laad icoon getoond worden terwijl tijdens de opstart procedure de app in geheugen wordt geladen en dus nog niets getoond kan worden.

Een performante app betekent in vele gevallen een snelle app, dus kan een trage opstartsnelheid leiden tot frustratie. Aangezien elke ontwikkelaar mikt op een zo goed mogelijke user experience (UX), is het interessant om ook de opstartsnelheid van een applicatie te onderzoeken. De vraag die hier beantwoord werd was volgende 'Wat is de impact van beide ontwikkelingstechnieken op de opstartsnelheid?'.

4.2.1 Opzet

Om het verschil in opstartsnelheid tussen beide platformen te testen, werd gebruik gemaakt van de Hello World applicaties. Door de kleine omvang van de applicaties, waren de verschillen in tijd beperkt. Ook werd rekening gehouden met het feit dat deze opstart

Tabel 4.5: Opstartsnelheid

	Android	**Flutter**
Hoogst	985ms	848ms
Laagst	582ms	293ms
Mediaan	649ms	416,5ms
Gemiddeld	681,31ms	454,38ms
Standaard afwijking	89,6450571ms	126,5891134ms

tijden variabel zijn. Daarom werd het onderzoek gevoerd aan de hand van een groot aantal iteraties. Hieruit kon een gemiddelde opstarttijd berekend worden per applicatie, die het mogelijk maakte om beide platformen met elkaar te vergelijken.

Hierbij dient wel vermeld te worden dat beide applicaties vanaf nul gestart werden. Het is namelijk zo dat applicaties uit drie verschillende toestanden gestart kunnen worden. De Engelse termen voor deze drie opstartprocedures zijn volgende: cold start, warm start en hot start. (zie appendix 10.2) In dit onderzoek werd gebruik gemaakt van de cold start procedure. Deze procedure is degene die de grootste uitdaging vormt in het kader van het minimaliseren van de opstarttijd.

In Android 4.4 (API-niveau 19) en hoger bevat logcat een regel uitvoer met de waarde 'Displayed'. Deze waarde vertegenwoordigt de hoeveelheid tijd die is verstreken tussen het starten van het proces en het voltooien van het tekenen van de bijbehorende activiteit op het scherm. Dit werd gebruikt voor het bepalen van de tijd.

In Flutter werd het commando flutter run –trace-startup –profile gebruikt. De trace-uitvoer wordt opgeslagen als een JSON-bestand met de naam start_up_info.json onder de build directory van het Flutter-project. De uitvoer geeft de verstreken tijd weer vanaf het opstarten van de app tot deze trace events. Echter toont de uitvoer van dit commando ook de tijd nodig voor Flutter om de app vanaf nul op te starten tot de eerste pixel op het scherm wordt getekend.

4.2.2 Resultaten

Zie tabel 4.5

4.3 Het CPU gebruik van de applicatie

Een Central Processing Unit of kortweg CPU is het als het ware het brein van een computer. Het is een stuk hardware dat aan de hand van een aantal basisoperaties, programmas uitvoert. Om deze operaties performant uit te voeren heeft de CPU echter nood aan een vaste, degelijke bron van energie. In het geval van mobiele apparaten, zoals een laptop of een smartphone, is deze energiebron de batterij van het toestel. Vandaag de dag moe-

ten, en zijn, smartphones vrij tot zeer compact. Dit heeft echter ook een nadeel. Door de compactheid van de toestellen, is de plek voor de batterij ook eerder beperkt. Om de capaciteit van de batterij zo goed mogelijk te benutten, moeten de processen die op de toestellen draaien dus eerder performant zijn. Hiermee wordt bedoeld dat ze energiezuinig moeten zijn zonder snelheid in te leveren.

Een app die veel CPU werk vereist zal dus meer energie verbruiken, de batterij sneller doen leeglopen en kan zorgen voor oververhitting van de batterij. Een probleem dat voorkomt wanneer de app meer resources gebruikt als nodig.

Aangezien het beheer van de batterijduur een dagelijkse ervaring is voor velen, wil men zo weinig mogelijk batterij verspelen aan korte app interacties. Gebruikers die merken dat een app te veel vraagt van de CPU zullen deze app dan ook sneller verwijderen. Daarom was het interessant om te onderzoeken of de onderliggende frameworks hier een aandeel in hebben.

4.3.1 Opzet

Voor het onderzoek naar CPU gebruik werden de onderzoeksapplicaties vergeleken. Deze applicaties zijn omvangrijk en bevatten meerdere features die CPU intensieve taken uitvoeren.

Het CPU gebruik wordt uitgedrukt in procent. Hoe hoger dit getal hoe meer CPU de app in gebruik nam. De gehele app werd doorlopen en het gemiddelde CPU gebruik werd vergeleken. Uitschieters werden ook opgenomen in de resultaten voor het maken van vergelijkingen omtrent CPU intensieve components.

Voor het onderzoek naar CPU gebruik werd voor Android gebruik gemaakt van de Android Profiler. Een krachtige monitoring tool die het mogelijk maakt om live CPU, geheugen, netwerk en batterij resources op te volgen. Meer specifiek werd gekeken naar de CPU Profiler. Voor Flutter werd gebruik gemaakt van de Flutter DevTools. Een uitgebreide tool die in Flutter 2.0 een grote update kende. Hiermee kunnen live zaken zoals logging, geheugen gebruik, CPU gebruik, netwerk requests worden opgevolgd.

4.3.2 Resultaten

Zie tabel 4.6

Tabel 4.6: Opstartsnelheid

	Android	**Flutter**
Hoogst	33.6%	35.4%
Laagst	1.0%	1.0%
Gemiddeld	12.7%	13.2%
Standaard afwijking	7.5146739%	8.1725839%

4.4 Conclusie

bij grotere Flutter APKs wordt het verschil klein en insignificant.

Wat uit deze resultaten kan geconcludeerd worden is dat een duidelijk verschil is tussen enerzijds de native APKs en de cross-platform APKs. Het is namelijk zo dat native APKs aanzienlijk kleiner zijn in vergelijking met hun cross-platform tegenhangers.

Het testen op CPU-verschillen in de applicaties toonde aan dat er niet veel verschil was in CPU-gebruik tussen native Android builds en Flutter builds. Dit zou het gevolg kunnen zijn van het gebruik van een kleine applicatie. Het experiment zou andere resultaten kunnen opleveren als dit deel van het onderzoek op een grotere applicatie werd uitgevoerd.

5. Asynchroon Werken

Bij het schrijven van code maakt de ontwikkelaar keuzes over de achterliggende uitvoeringsprocessen. Moeten stukken code parallel of sequentieel verlopen met andere stukken code. Deze keuze is vaak onbewust maar wel belangrijk voor de UX van een applicatie.

Het uitvoeren van een stuk code is een proces. Wanneer gekozen wordt om een stuk code synchroon of sequentieel te laten verlopen, wordt gewacht op de uitkomst van dit proces vooraleer een nieuw proces kan gestart worden. Hierbij wordt elk proces afgewerkt en enkel wanneer een bepaald proces afgewerkt is, kan een ander proces gestart worden.

Wanneer gekozen wordt om asynchroon of parallel te werken, worden processen niet eerst afgerond voor volgende processen kunnen beginnen. Het tweede proces in de wachtrij hoeft dan niet te wachten tot het eerste proces voltooid is alvorens te kunnen beginnen. Hierbij zijn dus meerdere processen op hetzelfde moment actief. Hier worden verschillende processen afgehandeld op verschillende threads. Een thread kan beschouwd worden als een rijstrook op de autosnelweg, er zijn verschillende threads waarop verschillende processen tegelijkertijd kunnen uitgevoerd worden, net zoals op de verschillende rijstroken meerdere autos tegelijkertijd kunnen rijden.

In de documentatie van Google over UI-threads en niet-UI-bewerkingen staat: Terwijl een schermupdate plaatsvindt, probeert het systeem om de 16 ms een blok UI code uit te voeren om soepel te renderen met 60 frames per seconde. Om het systeem dit doel te laten bereiken, moet de UI/View-hiërarchie worden bijgewerkt in de main thread. Als de wachtrij van de main thread echter taken bevat die te lang zijn om de update snel genoeg te voltooien, moet de app dit werk naar een andere thread verplaatsen. Als de main thead het uitvoeren van blokken code niet binnen 16 ms kan voltooien, kan de gebruiker haperingen, vertragingen of een gebrek aan UI-respons op invoer waarnemen. Als de main

thread gedurende ongeveer vijf seconden blokkeert, geeft het systeem een dialoogvenster Application Not Responding (ANR) weer, zodat de gebruiker de app direct kan sluiten.

Dus voor bepaalde taken zoals het ophalen en persisteren van data wordt vaak gekozen voor asynchrone taken. Op zowel Android als Flutter zijn verschillende manieren aanwezig om taken asynchroon uit te voeren. In dit hoofdstuk worden de meest gebruikte manieren aangehaald.

5.1 Opzet

Binnen asynchroon programmeren komen een aantal termen vaak terug. Zo zijn de vier belangrijkste concurrency, parallellisme, asynchrone taken en threads. Concurrency verwijst naar het beheren van meerdere uitvoeringsthreads, waarbij parallellisme meer specifiek is, meerdere uitvoeringsthreads die tegelijkertijd worden uitgevoerd. Concurrency is de bredere term die parallellisme kan omvatten. Asynchrone methoden zijn niet direct gerelateerd aan de vorige twee concepten, asynchroon wordt gebruikt om de indruk te wekken van gelijktijdige of parallelle taken, maar in feite wordt normaal gesproken een asynchrone methode aanroep gebruikt voor een proces dat werk moet doen buiten de huidige applicatie en de applicatie niet wilt geblokt worden in afwachting van het antwoord. Een thread is de kleinste opeenvolging van geprogrammeerde instructies die onafhankelijk van elkaar kunnen worden beheerd.

Toegang krijgen tot dezelfde gegevens vanuit meerdere threads, het juiste gedrag en goede prestaties behouden, is de echte uitdaging van *concurrent* programmeren. De beste gegevensstructuur die kan gebruikt worden om veilig gegevens over verschillende threads te delen, zijn wachtrijen. Threads communiceren meestal via wachtrijen en ze kunnen handelen als producent of consument. Een producent is een thread die informatie in de wachtrij plaatst, terwijl de consument degene is die ze leest en gebruikt. Een wachtrij kan gezien worden als een lijst waarin producenten gegevens aan het einde toevoegen, terwijl consumenten ze van het begin afnemen en lezen. Deze logica is wel beter bekend onder de naam FIFO (First In First Out). Threads plaatsen dus gegevens in de wachtrij. Deze gegevens worden ook wel berichten genoemd en zij bevatten de te delen informatie.

5.1.1 Android

In Android worden de grootste concepten bekeken rond asynchroon programmeren. Dit zijn niet alle opties, alleen de meest gebruikte. Hierbij worden simpele codevoorbeelden toegevoegd om inzicht te verwerven in de concepten.

5.1.2 Flutter

In Flutter worden 5 concepten bekeken. Eerst en vooral hebben we futures, de manier van asynchroon werken binnen Flutter. Hierbinnen wordt nog een onderscheid gemaakt tussen futures met en zonder callbacks. Vervolgens hebben we de widgets die samenwerken met deze futures namelijk StatefulWidget, StreamBuilder en FutureBuilder. Deze widgets maken het mogelijk om de UI aan te passen wanneer het resultaat van een future of stream. Ook hier zijn simpele codevoorbeelden toegevoegd.

5.2 Android

5.2.1 Callbacks

Callbacks zijn een programmeermechanisme. Het zijn functies die als argument aan een andere functie worden doorgegeven, die vervolgens binnen de buitenste functie wordt aangeroepen om een soort routine of actie te voltooien. Ze worden vooral gebruikt in samenwerking met één van de hieronder vermelde libraries. Het is mogelijk om deze te gebruiken zonder library maar dat is toegankelijk voor errors en moeilijk te onderhouden.

Sommige libraries bieden de mogelijkheid voor zowel synchrone (blokkerende) of asynchrone (niet-blokkerende) calls. Callbacks voorzien een manier om ze beide te doen, maar elke optie heeft zijn prijs. Als ervoor gekozen wordt om synchrone calls te gebruiken, moet je de threading-problemen zelf oplossen, maar zoals hiervoor vermeld is dit error prone. Als ervoor gekozen wordt om de asynchrone calls te gebruiken, moet niet aan de achtergrond threads gedacht worden, maar moet wel een soort van callback voorzien worden om een melding te krijgen wanneer de call geëindigd is.

Al wordt get aangeroepen van op de main thread, het zal een andere thread gebruiken om het verzoek uit te voeren. Zodra het resultaat beschikbaar is via het netwerk, wordt de callback aangeroepen op de main thread. Dit is een manier om langlopende taken af te handelen, en libraries zoals Retrofit kunnen u helpen netwerkverzoeken te doen zonder de main thread te blokkeren.

Callbacks hebben echter ook nadelen. In toepassingen op bedrijfsniveau is het vaak het geval dat er meerdere functies worden aangeroepen, die op de een of andere manier moeten worden verbonden of gecombineerd, waardoor de resultaten in complexere objecten moeten worden weergegeven. In deze gevallen wordt de code buitengewoon moeilijk om te schrijven, te onderhouden en te redeneren. Omdat het niet mogelijk is om een waarde van een callback terug te geven, moeten callbacks genest worden. Het is vergelijkbaar met het nesten van forEach of map functies. Elke bewerking heeft dan zijn eigen lambda-parameter. Bij het nesten van callbacks of lambda's krijgen we een groot aantal accolades, die elk een *local scope* vormen. Dit creëert op zijn beurt een structuur die indentatie- of callbackhel wordt genoemd.

Listing 5.1: Android Callback example

```
api.getSomething().enqueue(object: Callback<?> {
    override fun onResponse(
        call: Call<String>,
        response: Response<?>) {}

    override fun onFailure(
        call: Call<String>,
        t: Throwable) {}
})
```

5.2.2 AsyncTasks

AsyncTasks werd door Google deprecated gemaakt voor API level 30 en hoger. Deprecated of verouderd betekent dat Google aanbeveelt om naar iets anders over te stappen. Het betekent niet dat de klasse snel zal verwijderd worden uit Android. Aangezien AsyncTasks een veel gebruikte optie was, wordt deze opgenomen in het onderzoek. Het extenden van AsyncTask en het toevoegen van uw code in de doInBackground methode is alles wat nodig is om een lange berekening naar een andere thread over te dragen, en met de andere methoden kunt u veel eenvoudige use-cases afhandelen. Het bijhouden van de voortgang, het uitvoeren van een subtaak op de main thread voordat het hoofdwerk begint of nadat het is afgelopen, is net zo eenvoudig. Die eenvoud verklaart waarom zoveel mensen AsyncTasks gebruiken.

Er zijn enkele problemen met AsyncTask. De grootste is het gevaar op geheugenlekken. Zo moet de levenscyclus van de activiteit of het fragment in rekening gehouden worden. Daarom is het de verantwoordelijkheid van de programmeur om het AsyncTasks-gedrag af te handelen wanneer de activiteit wordt vernietigd. Dit betekent dat ze niet de beste optie zijn voor langdurige bewerkingen en ook, als de app zich op de achtergrond bevindt en de app wordt beëindigd door Android, wordt uw achtergrondverwerking ook beëindigd.

Als de use-cases vrij eenvoudig zijn en ervoor gezorgd wordt dat er geen enkele verwijzing naar de context of het view-object bewaart wordt(impliciet, expliciet, direct, indirect...), zou dat alles kunnen zijn wat nodig is om blocking tasks naar een achtergrond thread te delegeren. Het is zelfs mogelijk om ze te gebruiken met executors en pools van threads om de zaken een beetje schoner te houden.

Net als Callbacks werken AsyncTasks beter voor simpele taken. Complexe scenarios met verschillende uitvoeringsthreads en volgorde van bewerkingen, zullen snel de limieten bereiken.

Listing 5.2: Android AsyncTask example

```
class LongRunningOperation() : AsyncTask<Void, Void, String>() {
    override fun doInBackground(vararg params: Void?): String? {
        // Long running Task
    }

    override fun onPreExecute() {
        super.onPreExecute()
        //Before executing
    }

    override fun onPostExecute(result: String?) {
        super.onPostExecute(result)
        //After executing
    }
}
```

```
LongRunningOperation().execute(String);
```

5.2.3 Runnables, Threads, Handlers en AsyncTask

Een andere manier om taken in de achtergrond uit te voeren is door een Runnable te gebruiken die de code bevat die u op de achtergrond wilt uitvoeren, en die op een nieuwe thread uit te voeren. Als de gebruikersinterface achteraf moet bijgewerkt worden, kan dat niet gebeuren binnen de Runnable, omdat deze op een andere thread dan de UI thread draait. Deze Runnable kan doorgegeven worden aan een Handler of aan een Thread.

Thread
Als new Thread(runnable).start() gebruikt wordt, maakt deze een nieuwe thread aan voor de taak asynchroon uit te voeren. Een thread bevat een Looper. Wanneer de main thread wordt uitgevoerd, zal de Looper herhalen en de Runnables één voor één uitvoeren. Gebruik Thread dus als er zwaar werk moet gebeuren zoals bijvoorbeeld netwerkcommunicatie. Als er veel thread nodig is, heeft ExecutorService misschien meer de voorkeur. Het is mogelijk om een eigen threadmodel te maken zoals AsyncTask en Handler, maar dit vereist een goede kennis van Java's Multi-Threading implementatie. Het starten van threads om je Runnable tasks uit te voeren, ze laten wachten (wait) en samenkomen (join) is een benadering op een vrij laag niveau. Deze verouderde manier is ook vatbaar voor deadlocks en overzicht bewaren over de threads is ook niet altijd even simpel. Hierbij moet ook rekening gehouden worden met het feit dat testen en debuggen zo goed als onmogelijk zijn. Tegenwoordig bestaan er betere opties op Android.

Handler
Wanneer new Handler().post(runnable) gebruikt wordt, is het Runnable-object aan de Looper toegevoegd en wordt de code dus later in dezelfde thread uitgevoerd. Handler heeft de voorkeur wanneer UI-objecten uit een andere thread moeten bijgewerkt worden, dan is het noodzakelijk dat deze objecten alleen in de UI Thread kunnen worden bijgewerkt. Ook is het beter voor het uitvoeren van simpele code omdat het lichter en sneller is.

AsyncTask
Hiervoor werd AsyncTask reeds aangehaald. Toch kan het handig zijn om deze in de vergelijking toe te voegen. AsyncTask en Handler zijn geschreven in Java (intern gebruiken ze een Thread), dus alles wat we met Handler of AsyncTask kunnen doen, kunnen we ook bereiken met een Thread. De meest voor de hand liggende reden voor het gebruik van AsyncTask en Handler is de communicatie tussen de caller thread en de worker thread. Natuurlijk kunnen we op andere manieren communiceren tussen twee threads, maar er zijn veel nadelen (en gevaren) vanwege de veiligheid van de thread. Daarom is het best om Handler en AsyncTask te gebruiken. Deze klassen doen het meeste werk. Gebruik AsyncTask over Handler wanneer de caller thread een UI-thread is. Dit is wat Android documentatie zegt: AsyncTask maakt correct en gemakkelijk gebruik van de UI-thread mogelijk. Deze klasse maakt het mogelijk om achtergrondbewerkingen uit te voeren en resultaten op de UI-thread te publiceren zonder threads en/of handlers te hoeven manipuleren.

Listing 5.3: Android Runnable example

```
val runnable = Runnable {
    // Long running task
}

/*Example Handler*/
Handler().post(runnable)

/*Example Thread*/
Thread(runnable).start()
```

5.2.4 Kotlin coroutines

Kotlin-coroutines introduceren een nieuwe stijl van concurrency die op Android kan worden gebruikt om asynchrone code te vereenvoudigen. Coroutines zijn in versie 1.3 toegevoegd aan Kotlin. Ze zijn gebaseerd op gevestigde concepten uit andere talen. Een coroutine is een instantie van annuleerbare berekening. Het is conceptueel vergelijkbaar met een thread echter bieden Coroutines concurrency, maar geen parallellisme. Een coroutine is niet aan een bepaalde thread gebonden. Het kan de uitvoering ervan in de ene thread opschorten en in een andere thread hervatten. Coroutines kunnen worden gezien als lichte threads.

Ze bieden een goede oplossing op twee grote problemen.

- Langdurige taken: taken die te lang duren om de main thread te blokeren
- Main-safety: staat toe om elke suspend functie aan te roepen vanop de main thread

Coroutines bouwen verder op gewone functies door het toevoegen van twee nieuwe operaties. Bij de bestaande operaties invoke en return komen suspend en resume bij. suspend pauzeert de uitvoering van de huidige coroutine en slaat daarbij alle lokale variabelen op. resume zet een geschorste coroutine voort vanaf de plaats waar het was onderbroken

Er zijn verschillende manieren om met coroutines een stream van waarden terug te geven maar daar gaat dit onderzoek niet verder op in.

Listing 5.4: Android Coroutine example

```
fun main() = runBlocking {
    longRunningOperation()
}

suspend fun longRunningOperation() = coroutineScope {
    launch {
        //Long running task
    }
    //Other stuff
}
```

5.2.5 Reactive programming (RxJava)

In de afgelopen jaren is een andere mogelijkheid in een stroomversnelling gekomen, niet alleen voor Java, maar voor de meeste platforms. Reactive Extensions (ook bekend als Rx of ReactiveX) produceren libraries voor de meest voorkomende talen en impliceren een andere manier van denken om met asynchroon programmeren om te gaan. RxJava, dat net als elk Java-project op Android kan worden gebruikt, bevat een verzameling klassen en operators die u helpen bepaalde besturingsstromen om te keren zonder de controle over uw gegevensstromen en gebeurtenissen te verliezen.

Met behulp van de *Observables* en *Subscriptions* van RxJava is het mogelijk om een *Observer* te creëren die de methoden önNext", önErrorën önCompletedöverschrijft en te subscriben op een methode die een Observable retourneert. De Observable zendt items terug naar de Observer, waardoor de önNext-methode wordt aangeroepen met toegang tot het verzonden item. Als de Observable klaar is met het uitzenden van objecten, wordt önCompletedäangeroepen. Als er op enig moment een uitzondering wordt gegenereerd, wordt önErroräangeroepen met die uitzondering, wat een eenvoudige foutafhandeling mogelijk maakt. De enige opkuis die moet gebeuren is in onDestroy-methode te unsubscriben voor geheugenlekken te voorkomen.

Het hele concept berust op een combinatie van de klassieke Observer- en Iterator-patronen. Dit lijkt misschien een andere callback-aanpak, maar Rx is eigenlijk veel krachtiger dan dat: de grootste kracht ligt in het feit dat het een gelijkenis toont met functioneel programmeren. Dit blijkt duidelijk uit het gebruik van de meerdere operators die beschikbaar zijn in de RxJava-bibliotheek. Er zijn tientallen, zo niet honderden operators waarmee u uw gegevens kunt transformeren, uw streams kunt combineren of synchroniseren, tijdsbewerkingen kunt afhandelen, van thread kunt wisselen, enz. Met deze enorme toolbox is het koppelen van twee calls net zo eenvoudig als het gebruik van flatmap, het manipuleren van threads wordt gedaan met observeOn of subscribeOn, collecties kunnen worden samengevoegd met een zip en gegevens kunnen worden toegewezen aan meer geschikte types om aan de voorkeur te voldoen. Complexe algoritmen met meerdere asynchrone bewerkingen zijn veel leesbaarder. Als laatste is het mogelijk om met Rx-programmering asynchrone code te schrijven op een testbare manier. Zelfs als de code tijdsbewerkingen met zich meebrengt (vertragingen, time-outs ...), kun je test schedulers gebruiken om tijd te simuleren en te controleren of je code zich in milliseconden correct gedraagt.

Een nadeel dat Reactive programming met zich meebrengt is de stijle leercurve. Het kan een overweldigende taak zijn om de opbouw van een blok asynchrone code aan te passen om het meeste eruit te halen. Indien een eenmalige oplossing gezocht wordt voor een eenvoudig use case, wordt reactive niet aangeraden. Wanneer er vaak asynchrone code moet geschreven worden voor complexe use cases kan het interessant zijn om zich in reactive te verdiepen.

Listing 5.5: Android RxJava example

```
val observable = PublishSubject.create<Int>()

observable
```

```
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe({ value ->
        //do something with the value if success
}, {
        //do something with value if failure
})
```

5.3 Flutter

Dart is een single threaded taal die gebruikmaakt van event loops om asynchrone taken uit te voeren. Het kan associëren met verschillende codes die in discrete threads worden uitgevoerd. De bouwmethode in Flutter is echter synchroon. Het gebruik van synchrone code in Dart kan vertragingen veroorzaken en de uitvoering van het hele programma blokkeren. Asynchrone programmering pakt dit probleem aan. Bovendien zorgt dit voor een verbeterde uitvoering van de applicatie en een verbeterde reactiesnelheid van de applicatie

Dart event loop
Zodra een app geopend wordt, vinden verschillende events plaats in onvoorspelbare volgorde, totdat de app gesloten wordt. Elke keer een event plaatsvindt, komt deze in een wachtrij en wacht deze om verwerkt te worden. De Dart event loop haalt het event bovenaan de wachtrij op, verwerkt deze en activeert een callback totdat alle gebeurtenissen in de wachtrij zijn voltooid.

De klassen Future en Stream en de trefwoorden async en wait in Dart zijn gebaseerd op deze eenvoudige event loop, waardoor asynchroon programmeren mogelijk wordt.

Futures
Een future (kleine letter "f") is een instantie van de klasse Future. Een Future in Flutter maakt het mogelijk om de io te beheren zonder zicht te moeten druk maken over threads of deadlocks. Een future vertegenwoordigt het resultaat van een asynchrone bewerking en kan twee statussen hebben: onvoltooid of voltooid.

- Onvoltooid: Wanneer een asynchrone functie aangeroepen wordt, retourneert deze een onvoltooide future. Deze future wacht tot de asynchrone bewerking van de functie voltooid is of een fout genereerd.
- Voltooid: Als de asynchrone bewerking slaagt, wordt de toekomst voltooid met een waarde. Anders wordt het voltooid met een fout.

Future met callback
Een van de meest voorkomende Use Cases voor asynchroon programmeren is het verkrijgen van gegevens via een netwerk, zoals via een REST-service met HTTP.

Listing 5.6: Flutter callback example

```
http.get("https://example.com").then((response) {
```

```
    if (response.statusCode == 200) {
        //do something with the response
    }else {
        //do something else
    }
});
```

Het codevoorbeeld toont het klassieke patroon voor het consumeren van futures. De aanroep naar http.get() retourneert een onvolledige future. Onthoud dat het verkrijgen van resultaten via HTTP tijd kost, en het is ongewenst dat de app niet reageert in deze periode. Daarom wordt de future meteen terug gehaald en wordt verder gegaan met het uitvoeren van de andere regels code. Die andere regels gebruiken de methode then() van de Future-instantie om een callback te registreren die wordt uitgevoerd wanneer het antwoord binnenkomt.

async en await
Dart biedt een alternatief patroon voor het maken van asynchrone calls, een patroon dat meer op gewone synchrone code lijkt, waardoor het gemakkelijker te lezen en te volgen is. De async/wait-syntaxis regelt veel van de logistiek van futures voor u:

Dart voorziet twee keywords die hierbij kunnen gebruikt worden:

- async: wordt geplaatst voor de braces van de functie om deze als asynchroon te markeren.
- wait: wordt geplaatst voor de methode aanroep van een async methode. Het wait-sleutelwoord werkt alleen binnen een asynchrone functie.

Een asynchrone functie wordt synchroon uitgevoerd tot het eerste wait-sleutelwoord. Dit betekent dat binnen een asynchrone functie alle synchrone code vóór het eerste wait-sleutelwoord onmiddellijk wordt uitgevoerd.

Listing 5.7: Flutter async example

```
Future <?> longRunningOperation () async {
    //Long running task
}

void otherFunction() async {
    return await longRunningOperation().then((value){
        //do something with the value
    }, onError: (error){
        //do something with the error
    });
}
```

Widgets opbouwen die gebruik maken van futures is waar de moeilijkheid ligt in Flutter. Dit kan aan de hand van één van volgende manieren.

StatefullWidget
Dit is een eenvoudige manier die best gebruikt voor simpele Use Cases. Wanneer de asynchrone data binnenkomt, kan een rebuild getriggerd worden aan de hand van setState().

Listing 5.8: Flutter StatefulWidget example

```
class _MyFutureWidgetState extends State<MyFutureWidget> {
    String? value;

    @override
    void initState() {
        super.initState();
        fetchFuture().then((result) {
            setState(() {
                value = result;
            });
        });
    }

    @override
    Widget build(BuildContext context) {
        return Container(
                    child: (() {
                        if (value == null) {
                            //return waiting widget
                        }
                    //return widget showing success
                    })()
        )
    }
}

Future<?> fetchFuture() async {
    //Long running task
}
```

FutureBuilder
FutureBuilder biedt een mooiere, betere manier aan om met futures in Flutter om te gaan. FutureBuilder maakt het mogelijk widgets te bouwen wanneer deze te maken hebben met een future. Met deze widget is het mogelijk om de huidige status van de future op te vragen. Zo is het mogelijk om verschillende widgets te tonen wanneer de data wordt geladen en wanneer deze beschikbaar is. Zoals de naam zegt, bestaat deze widget uit twee grote delen, een future en een builder. De builder heeft een context en snapshot. Het snapshot is een onveranderlijke weergave van de meest recente interactie met een asynchrone berekening. Het heeft meerdere eigenschappen. Wanneer een asynchrone berekening plaatsvindt, is het handig om de status van de huidige verbinding te kennen, wat mogelijk is via snapshot.connectionState.

De connectionState heeft vier gebruikelijke stromen:

- none: initiële gegevens (wanneer ingesteld)
- waiting: de asynchrone bewerking is begonnen. De gegevens zijn meestal null
- active: gegevens zijn niet-null en kunnen in de loop van de tijd veranderen
- done: gegevens zijn niet-null

snapshot.data retourneert de nieuwste gegevens en snapshot.error retourneert het nieuwste error object. snapshot.hasData en snapshot.hasError zijn twee handige getters die controleren of er een fout of data is ontvangen.

Listing 5.9: Flutter FutureBuilder example

```
FutureBuilder(
    future: functionThatReturnsFuture(),
    builder: (BuildContext context, AsyncSnapshot snapshot) {
        if (snapshot.hasData) {
            //return widget showing success
        } else {
            //retur nwaiting widget
        }
    },
),
```

StreamBuilder

StreamBuilder en FutureBuilder zijn bijna identieke concepten. StreamBuilder levert echter periodiek data, dus moet er vaker naar geluisterd worden dan dat het geval is bij FutureBuilder, waar één keer luisteren. De StreamBuilder-widget subscribes en unsubscribes automatisch op een stream. Wanneer een widget wordt gedisposed, moet men zich geen zorgen maken over het unsubscriben, en het is net bij dit dat een geheugenlek zou kunnen onstaan. StreamBuilder zorgt er dus met andere woorden voor dat geheugenlekker quasi niet kunnen voorkomen, wat een groot voordeel is.

Listing 5.10: Flutter Stream example

```
StreamBuilder(
    stream: functionThatReturnsStream(),
    builder: (BuildContext context, AsyncSnapshot snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
            //return waiting widget
        } else if (snapshot.connectionState == ConnectionState.done) {
            //return Widget showing success
        } else {
            //return Widget showing failure
        }
    },
),
```

5.4 Conclusion

Het is duidelijk dat Android, het meer ontwikkelde platform, ook meer opties aanbiedt. Dit maakt het platform toegankelijker voor programmeurs met verschillende achtergronden. Zij voelen zich sneller thuis in de gekende concepten. Aan de andere kant, als men nieuw is in het vak kan het aantal opties voor asynchroon werk nogal overweldigend over komen. Hoewel er meerdere manieren zijn om hetzelfde te bereiken . Flutter biedt één overzichtelijke manier om asynchroon programmeren aan te pakken. Het is duidelijk en gemakkelijk om aan te leren. De integratie van futures en streams met widgets is duidelijk gedocumenteerd.

6. Gebruik van online APIs

Het developer vakjargon bevat een aantal veel voorkomende termen, één van deze is API. Een Application Programming Interface, of dus kortweg API, is software die het toelaat om verschillende applicaties of applicatielagen met elkaar te laten communiceren. Een van de primaire use cases waarvoor een API gebruikt wordt in de mobiele applicatie wereld, is die om de zogenaamde API calls uit te voeren. Het doel van deze API calls zijn om vanuit de mobiele applicatie, de API aan te spreken om data op te halen die vervolgens verwerkt en getoond kan worden. Een API zorgt er in dit geval voor dat een mobiele applicatie in connectie staat met de cloud. Naast het ophalen van data kan er vaak ook nieuwe data naar de API gestuurd worden, of kan bestaande data aangepast worden.

Wanneer een app gebruik wilt maken van data die niet door de eindgebruiker is ingegeven moet een netwerk connectie tot stand worden gebracht. Daarom is het voor elke nieuwe app ontwikkelaar belangrijk om op de hoogte te zijn van de werking hiervan.

6.1 Opzet

Voor dit deel van het onderzoek werd gebruik gemaakt van een online API en de al dan niet ingebouwde platform specifieke functionaliteiten voor het aanspreken van deze services. In de onderzoeksapplicatie werden verschillende API calls gedaan die later in dit hoofdstuk worden beschreven.

Eén van de bekendere en meest gebruiksvriendelijke APIs op het web is de PokéApi. Deze RESTful API, die informatie biedt omtrent allerlei Pokémon, is goed ondersteunt en duidelijk gedocumenteerd. Deze documentatie is te lezen op een overzichtelijke site.

Tijdens het uitvoeren van dit experiment, werd gebruik gemaakt van zogenaamde data klassen. Deze klassen worden onder andere gebruikt om de verschillende objecten op te bouwen die terugkomen van de API.

6.2 Android

De meeste netwerk geconnecteerde apps maken gebruik van HTTP om data te versturen en te ontvangen over het internet. Android voorziet hiervoor twee clients: HttpURLConnection en Apache HTTP Client. Daarbuiten kunnen ook een aantal goed ontwikkelde frameworks gebruikt worden. Enkele van de libraries die hiervoor in Android makkelijk te gebruiken zijn, zijn OKHttp, Volley, Retrofit, Picasso, Fresco.

Googles HTTP clients:
In de begin jaren van Android waren juist twee HTTP Clients beschikbaar. Echter waren beide niet goed geïntegreerd in het platform. HttpUrlConnection had last van enkele bugs terwijl Apache Clients niet meer werd geüpdatet. Terwijl Google ervoor koos om Apache-based AndroidHttpClient later deprecated te maken besloten ze om de bugs in HttpURLConnection wel op te lossen.

Binnen Android is dit een vlugge gemakkelijke manier om een netwerk request uit te voeren. Echter wordt deze al snel ingewikkeld als deze verder uitbreidt.

Listing 6.1: Android HttpUrlCpnnection example

```
val url = URL("http://www.android.com/")
val urlConnection: HttpURLConnection
    = url.openConnection() as HttpURLConnection
try {
    val inp: InputStream
        = BufferedInputStream(urlConnection.getInputStream())
    //Do something with response
} finally {
    urlConnection.disconnect()
}
```

OKHttp:
Google besloot uiteindelijk om de goed ondersteunde OKHttp library op te nemen in Android. Hierdoor konden ze, zoals hiervoor vermeld, de Apache library deprecated maken. Ondertussen heeft de OKHttp library al een heleboel extra features die netwerken zoveel gemakkelijker maken. Zo voorziet de library een goede oplossing voor asynchrone requests die sinds Android 3.0 op een aparte thread moeten worden uitgevoerd. Om vergelijkbare logica met HttpUrlConnection te implementeren, zou een veel omvangrijkere implementatie moeten worden bedacht om AsyncTask-wrapper of afzonderlijke threads te gebruiken. Het wordt nog ingewikkelder als u functies zoals cancelation en connection pooling moet ondersteunen.

Listing 6.2: Android OKHttp example

```
val client = OkHttpClient()
val url = URL("http://www.android.com/")

fun run() {
    val request = Request.Builder()
    .url(url)
    .build()

    client.newCall(request).execute().use { response ->
        if (!response.isSuccessful)
            throw IOException("Unexpected code $response")
        //Do something with response
    }
}
```

Volley:
Na het zien van de snelle toename in populariteit van OKHttp, besloot Google om Volley te lanceren. Alles in Volley is voortgebouwd op HttpUrlConnection. Ondanks de late release van de library, heeft deze zijn voor- en nadelen. Zo voorziet volley automatische planning van netwerkverzoeken, meerdere gelijktijdige netwerkverbindingen, ondersteuning voor prioritering van verzoeken maar maakt de library bijvoorbeeld gebruik van een hardcoded networking thread pool size.

Listing 6.3: Android Volley example

```
val queue = Volley.newRequestQueue(this)
val url = "http://www.android.com/"

val stringRequest = StringRequest(Request.Method.GET, url,
Response.Listener<String> { response ->
    //Do something with response
},
Response.ErrorListener {
    //Do something with error
})

// Add the request to the RequestQueue.
queue.add(stringRequest)
```

Retrofit (square):
Retrofit is een speciale HTTP-client voor Android en Java. De type-safe client maakt verbinding met een REST-webservice door de API te vertalen naar Java-interfaces. Deze krachtige bibliotheek maakt het gemakkelijk om JSON- of XML-gegevens te gebruiken, parsed in Plain Old Java Objects. De open-source Retrofit heeft zijn basis bovenop enkele andere krachtige bibliotheken en tools. Het maakt gebruik van OKHttp om netwerkverzoeken af te handelen.

Listing 6.4: Android Retrofit example

```
data class SomeData(var property: String)

interface ApiInterface {
    @GET("/")
    fun get() : Call<List<SomeData>>

    companion object {
        val url = "..."
        fun create() : ApiInterface {
            val retrofit = Retrofit.Builder()
            .addConverterFactory(GsonConverterFactory.create())
            .baseUrl(url)
            .build()
            return retrofit.create(ApiInterface::class.java)
        }
    }
}

val apiInterface = ApiInterface.create().get()

apiInterface.enqueue( object : Callback<List<SomeData>>{
    override fun onResponse(call: Call<List<SomeData>>?, response: Resp
        if(response?.body() != null)
            //Do something with result
    }
    override fun onFailure(call: Call<List<Movie>>?, t: Throwable?) {
        //Do something with error
    }
})
```

Picasso (square):
Picasso is specifiek voor afbeeldingen, met een sterke HTTP downloading en caching library. Zowel Picasso als Retrofit kunnen worden geconfigureerd om OkHttpClient als de standaard HTTP-client te gebruiken.

Listing 6.5: Android Picasso example

```
val imageUri = "https://...jpg"
val imageView = findViewById<ImageView>(R.id.imageId)
Picasso.get().load(imageUri).into(imageView)
```

Glide (bumptech):
Glide is een snel en efficiënt open source-framework voor mediabeheer en het laden van afbeeldingen voor Android dat mediadecodering, geheugen- en schijfcaching en resourcepooling verpakt in een eenvoudige en gebruiksvriendelijke interface. Deze library is vergelijkbaar met Picasso, maar voorziet een aantal extra features zoals GIF animaties,

thumbnail generatie en still videos. Glide gebruikt standaard een aangepaste op HttpUrl-Connection gebaseerde stack, maar bevat in plaats daarvan ook utility libraries die kunnen worden verbonden met het Volley of OkHttp.

Listing 6.6: Android Glide example

```
val url = "https://...jpg"
Glide.with(itemView)
.load(url)
.centerCrop()
.placeholder(R.drawable.ic_image_place_holder)
.error(R.drawable.ic_broken_image)
.fallback(R.drawable.ic_no_image)
.into(itemView.ivPhoto)
```

Fresco (facebook):
Fresco is een krachtig systeem voor het weergeven van afbeeldingen in Android-applicaties. Fresco zorgt voor het laden en weergeven van afbeeldingen. Het laadt afbeeldingen van het netwerk, lokale opslag of lokale bronnen en geeft een tijdelijke aanduiding weer totdat de afbeelding is aangekomen. Het heeft twee cacheniveaus; een in het geheugen en een andere in de interne opslag. In Android 4.x en lager plaatst Fresco afbeeldingen in een speciale regio van Android-geheugen. Hierdoor kan uw toepassing sneller werken - en veel minder vaak last hebben van de gevreesde OutOfMemoryError.

Listing 6.7: Android Glide example

```
Fresco.initialize(this)

val imageView = findViewById<SimpleDraweeView>(R.id.posterImage);
Uri uri = Uri.parse("https://...jpg");
imageView.setImageURI(uri);
```

Moshi(square):
Moshi is een moderne JSON-bibliotheek voor Android en Java. Het maakt het gemakkelijk om JSON in Java-objecten om te zetten en het kan even gemakkelijk Java-objecten serialiseren naar JSON. Moshi heeft ingebouwde ondersteuning voor het lezen en schrijven van de belangrijkste gegevenstypen van Java: int, float, char... maar ook Arrays, Collections, Lists, Sets, Maps, Strings en Enums. Het ondersteunt uw modelklassen door ze veld voor veld uit te schrijven.

6.3 Flutter

Voor Flutter wordt vooral gebruik gemaakt van de HTTP Package library.

http
Deze package, uitgebracht door dart, bevat een reeks functies en klassen op hoog niveau

die het gemakkelijk maken om HTTP-bronnen te gebruiken. Het is multi-platform en ondersteunt mobiel, desktop en de browser.

Listing 6.8: Flutter http example

```
import 'package:http/http.dart' as http;
var result = await http.get(Uri.parse('https://flutterdevs.com'));
```

dio
Een krachtige Http-client voor Dart, die ondersteuning biedt voor interceptors, algemene configuratie, FormData, annulering van aanvragen, downloaden van bestanden, time-out enz.

Listing 6.9: Flutter dio example

```
import 'package:dio/dio.dart';
void getHttp() async {
    try {
        var response = await Dio().get('http://www.google.com');
        print(response);
    } catch (e) {
        print(e);
    }
}
```

cached_network_image
Een Flutter library om afbeeldingen van het internet weer te geven en in de cachemap te bewaren. De CachedNetworkImage kan direct of via de ImageProvider worden gebruikt. Het bevat momenteel geen caching. Dit is een library die niet werd uitgebracht door Flutter of Dart maar is zeer populair onder de community en wordt goed onderhouden.

Listing 6.10: Flutter http example

```
CachedNetworkImage(
imageUrl: "http://via.placeholder.com/350x150",
placeholder: (context, url) => CircularProgressIndicator(),
errorWidget: (context, url, error) => Icon(Icons.error),
),

Image(image: CachedNetworkImageProvider(url))
```

6.4 Conclusie

Zoals in vorige hoofdstukken ook het geval was is ook hier Android veel rijker aan keuzes. Natuurlijk is het

7. App veiligheid

Security of beveiliging is vandaag de dag meer dan enkel een hot topic. Het is een nood en het moet een garantie zijn die aangeboden wordt aan de gebruiker. Doorheen de gehele applicatie is security een belangrijk aspect. Daarom is app beveiliging één van de grootste zorgen voor ontwikkelaars. Door een applicatie op een veilige manier te bouwen helpt dit de gebruiker meer vertrouwen te plaatsen in deze applicatie en wordt de apparaat integriteit behouden.

In dit hoofdstuk worden een aantal best practices aangehaald. Het is niet mogelijk om ze allemaal te noemen, dus werd een lijst gemaakt van de belangrijkste. Hierbij wordt ook vermeld hoe deze best practices geïmplementeerd worden op zowel native Android als Flutter apps.

7.1 Best-practices

Bewaar gegevens op een veilige manier
De persoonlijke informatie van de veilige opslaggebruiker mag alleen op het apparaat blijven staan als dat nodig is, en alleen toegankelijk worden gemaakt voor geautoriseerde gebruikers en processen. API-toegangstokens moeten bijvoorbeeld veilig op het apparaat worden opgeslagen. Android Keystore biedt de mogelijkheid om cryptografische sleutels in een container op te slaan, waardoor het moeilijker wordt voor ongeautoriseerde toegang.

Ter vergelijking: de plug-in flutter_secure_storage biedt veilige opslag via Android KeyStore. Voor een permanente opslag van eenvoudige gegevens is de plugin shared_preferences beschikbaar. Echter, citerend uit de officiële documentatie, kan geen van beide platformen garanderen dat schrijfbewerkingen na terugkeer op schijf blijven staan; daarom mag de plug-in niet worden gebruikt voor het opslaan van kritieke gegevens. Ontwikkelaars kunnen besluiten om Flutter-platformkanalen te gebruiken om platform specifieke API's aan te roepen voor hun behoeften.

Houd SDK, services en dependencies up-to-date
De meeste apps gebruiken externe bibliotheken en apparaat informatie om gespecialiseerde taken uit te voeren. Door de afhankelijkheden van de app up-to-date te houden, worden deze communicatiepunten veiliger gemaakt.

Vraag alleen persmissie voor wat je nodig hebt
Altijd om toestemming vragen wanneer het nodig is. Vraag geen toestemming vanaf het begin van de app en vraag alleen naar wat nodig is. Sommige apps kunnen bijvoorbeeld vragen om toegang tot de GPS-locatie, hoewel geen van de functies dit vereist. Dit is een slechte manier van werken voor twee redenen. Ten eerste worden onnodige privégegevens gegevens verzamelt. Ten tweede kunnen hackers mogelijk ook toegang krijgen tot deze gegevens als deze niet correct worden geïmplementeerd.

In Android worden permissies gevraagd aan de hand van een klein stuk code dat onder andere checkt of de permissie al is goedgekeurd of niet. Daarbij moet elke permissie in het manifest van de app worden vermeld.

In Flutter kunnen permissies worden gevraagd aan de hand van method channels of de permission_handler plugin.

Preventie van snapshots op de achtergrond
Apps die financiële informatie presenteren of betalingsfunctionaliteiten bieden, vereisen een hoger niveau van gegevensprivacy. Wanneer een app naar de achtergrond gaat, maakt het besturingssysteem een momentopname van de laatst zichtbare status om in de taakwisselaar te presenteren. Het is daarom gewenst om te voorkomen dat rekeningsaldi en betalingsgegevens worden vastgelegd door snapshots op de achtergrond.

Voor Android behandelt de FLAG_SECURE-vlag de inhoud van het venster als veilig, waardoor het niet in schermafbeeldingen verschijnt.

Omdat het voorkomen van snapshots op de achtergrond nauw verbonden is met de levenscyclus van een applicatie, biedt Flutter geen plug-in voor het voorkomen van snapshots. Om dit te bereiken moeten ontwikkelaars vertrouwen op native API's.

Detectie van gejailbreakte en geroote apparaten
Bij gejailbreakte apparaten worden hun ingebouwde beveiligingsmaatregelen ondermijnd, wat kwetsbaarheden introduceert en de persoonlijke informatie en inloggegevens van de gebruiker in gevaar kan brengen. Apps die persoonlijke informatie verwerken, mogen niet draaien op gejailbreakte apparaten; jailbreak-detectie is dus verplicht voor veel consumenten-apps.

Jailbreak-detectie omvat het scannen op verdachte mappen, bestanden en codefragmenten, wat platformspecifiek is. Flutter's SDK ondersteunt deze functie niet, maar bibliotheken van derden die native code inpakken om taken uit te voeren zoals flutter_jailbreak_detection en root_checker zijn beschikbaar. Toch zou het, in vergelijking met het gebruik van bibliotheken van derden, voor meer flexibiliteit en vertrouwen beter zijn om rechtstreeks het native framework te implementeren.

Code verduistering
Statische strings en tekstbestanden in een app package zijn relatief eenvoudig te bemachtigen en kunnen gevoelige informatie bevatten, zoals een API-sleutel. Een goed softwareontwerp minimaliseert de kans op het lekken van gevoelige informatie. Mocht het onvermijdelijk worden om informatie op een dergelijke manier te verwerken in de app, dan moet code verduistering worden gebruikt om de informatie door elkaar te gooien.

Code verduistering is een proces dat een uitvoerbaar bestand wijzigt of functie- en klasse namen verbergt, waardoor het moeilijk is om reverse-engineering toe te passen. Het beschermt het intellectuele eigendom van de klant en voorkomt ongeautoriseerde toegang, diefstal van inloggegevens of beveiligingsproblemen. Meestal ontwikkeld met native build-tools zoals het verwijderen van debugger-symbolen, code-optimalisatie en in-code verduisteringstechnieken.

Dart ondersteunt codeverduistering, maar is nog niet grondig getest. Flutter verkleint de Androidhosts niet. Daarom is het de verantwoordelijkheid van de ontwikkelaar om indien nodig native Android applicaties te obfusceren. Android ProGuard is in dit geval een handig hulpmiddel.

Encryptie-API
Afgezien van direct beschikbare API's voor veilige verbinding en opslag, willen ontwikkelaars soms codering op de applicatielaag toepassen om de gegevensbeveiliging te verbeteren. Aan de Flutter-kant bieden de crypto- en versleutelbibliotheken van Dart een reeks cryptografische hashfuncties met basisversleutelingsfuncties. Hoewel ze niet zo uitgebreid zijn als native Android-frameworks, moeten deze bibliotheken voldoen aan de basisvereisten voor gegevenscodering.

Verbindingsbeveiliging
Apps die gegevens via internet verzenden en ontvangen, moeten gevoelige informatie

beschermen tegen afluisteren en onbevoegde toegang. HTTPS, bekend als HTTP over TLS (Transport Layer Security), wordt vaak gebruikt om mobiel app-verkeer over uit te voeren vanwege de voordelen van gegevenscodering en authenticatie. Naast standaard TLS-configuraties kunnen functies zoals het vastzetten van certificaten en wederzijdse authenticatie de verbindingsbeveiliging verder verbeteren. Slecht geconfigureerde TLS-beveiligingsparameters, kunnen de verbinding kwetsbaar maken voor man-in-the-middle-aanvallen.

De functie Network Security Configuration op Android stelt ontwikkelaars in staat zich af te melden voor cleartext-verkeer, dwingt het gebruik van HTTPS af en voorkomt dat de app niet-versleutelde HTTP-verbindingen krijgt die worden geleverd door backend-servers.

De dart:io bibliotheek van Flutter ondersteunt HTTPS-verbinding met TLS-certificaten en de HttpClient-klasse kan worden gebruikt om HTTPS-verzoeken te doen met een aangepast vertrouwd certificaat dat wordt beheerd door SecurityContext-objecten. API-calls in Flutter kunnen dus net zo veilig zijn als vanuit native frameworks.

Lokale authenticatie
Biometrische authenticatie is geen nieuwkomer op het gebied van mobiele technologie. Het vereist dat gebruikers biologische kenmerken zoals vingerafdruk, gezicht of iris verifiëren om een gebruikersstroom te laten doorgaan. Een goed ontworpen gebruikersstroom met biometrische authenticatie verbetert de bescherming van de gevoelige informatie van de gebruiker aanzienlijk. Flutter biedt de plug-in local_auth om middelen te bieden om lokale authenticatie op het apparaat van de gebruiker uit te voeren. In vergelijking met native frameworks biedt local_auth meer generieke biometrische authenticatiefunctionaliteiten die voldoen aan de basisbehoeften. Om een volledig aangepaste gebruikerservaring voor biometrische authenticatie te bouwen, is het native framework nog de beste keuze.

7.2 Conclusie

Hoe complex de app ook is en welk ontwikkelingsframework ook gekozen werd, beveiliging is de grootste zorg voor elke ontwikkelaar. Het is dus de moeite waard om wat tijd te besteden aan het lezen van veelvoorkomende maar belangrijke beveiligingsrisico's van apps en hoe deze kunnen aangepakt worden. Aangezien het gebruik van apps zal blijven toenemen, zal dit ook gelden voor de gevaren. De ontwikkelaar is dan verantwoordelijk om de gebruiker gerust te stellen door de beveiliging van de app te verhogen en ervoor te zorgen dat informatie veilig blijft. Het niet slagen van de beveiligingstesten zal leiden tot gebruikers die overstappen naar de concurrentie.

Hoewel de beveiligingsopties van de native platformen talrijk zijn, moeten ontwikkelaars dezelfde mate van beveiliging kunnen bereiken met een combinatie van Flutter's framework, plug-ins en channeling naar native API's. Op dit moment vereist het ontwikkelen van Flutter apps nog steeds afzonderlijke codebases voor Android en iOS. Het combineren van rijke Flutter UI-widgets met krachtige native frameworks heeft echter een voordeel: het stelt ontwikkelaars in staat om krachtige native applicaties voor beide platforms te bouwen zonder onder te doen aan de applicatiebeveiliging. In combinatie met native frameworks heeft Flutter het potentieel en de mogelijkheid om consumenten apps van hoge kwaliteit te leveren. Google heeft het Flutter-framework gebouwd met alle beveiligingsproblemen en -fouten in gedachten. Flutter heeft bijna al de antwoorden op de meest voorkomende beveiligingsuitdagingen voor moderne apps.

8. Code Complexiteit

In dit hoofdstuk werd gekeken naar de complexiteit van de geschreven code. Dit schetst een beeld van de toegankelijkheid van de taal. Code complexiteit gaat in vele gevallen gepaard met het aantal lijnen code. Eerst en vooral is het niet altijd beter om alles in één beknopte lijn code te schrijven. Het is zo dat bij het schrijven van applicaties, vaak samengewerkt wordt met anderen aan dezelfde code. Complexe code is moeilijker om lezen en kan soms verkeerd geïnterpreteerd worden. Het is dus niet verkeerd om soms een extra lijn code te schrijven voor de complexiteit te verminderen. Echter moet hier een gezonde middenweg in gezocht worden aangezien veel lijnen code ook voor een complexe codebase kan zorgen.

8.1 Opzet

Er zijn verschillende manieren om de code complexiteit van een stuk code te bepalen. Veel van deze, hoewel ze een goede weergave van complexiteit opleveren, lenen zich niet voor gemakkelijke meting. Enkele van de meest gebruikte statistieken zijn:

- Cyclomatic complexity
- Halstead complexity measures
- Essential complexity

Echter werd in dit onderzoek geen gebruik gemaakt van deze algoritmes.

Voor dit deel van het onderzoek werd een notificatie lijst gemaakt. De opbouw van de code, het aantal lijnen code, de

Figuur 8.1: Android activity code snippet voor het maken van een notificatielijst.

```
package com.example.bap_experiment_android.ui.hoofdstuk7.activity

import android.content.Context
import android.content.Intent
import android.os.Bundle
import android.widget.ArrayAdapter
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.ViewModelProvider
import kotlinx.android.synthetic.main.activity_hoofdstuk7.*
import com.example.bap_experiment_android.R
import com.example.bap_experiment_android.ui.hoofdstuk7.viewmodel.Hoofdstuk7ViewModel


class Hoofdstuk7Activity : AppCompatActivity() {

    private lateinit var viewModel: Hoofdstuk7ViewModel

    companion object {

        fun getIntent(context: Context): Intent {
            return Intent(context, Hoofdstuk7Activity::class.java)
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_hoofdstuk7)

        val viewModelFactory = Hoofdstuk7ViewModel.Factory()
        viewModel = ViewModelProvider( owner: this, viewModelFactory).get(Hoofdstuk7ViewModel::class.java)

        val list = Array<String>( size: 20) { _ -> "notification" }
        notifications.adapter = ArrayAdapter<String>( context: this, R.layout.list_item_notification, list)
    }
}
```

8.2 Resultaten

Als we naar de resultaten als geheel kijken, wint Flutter de meeste categorieën in het ontwikkelingsgebied. Er zijn echter enkele verschillen die interessant zijn om op te letten bij het vergelijken van Flutter met native builds.

Code size
Zoals te zien is in figuur 8, had Flutter het minste aantal coderegels en bestanden die nodig waren om de applicatie te maken. Native iOS had de laagste grootte van projectbestanden en app-grootte, maar een aanzienlijk groter aantal coderegels dan de andere builds. De native Android had het meeste aantal bestanden gemaakt en vereiste minder coderegels dan de native iOS.

Code complexity
Voor dit deel v Het werd met name gekozen omdat het de meeste code bevatte die werd geschreven en omdat de andere weergave alleen een afbeelding en een titel bevatte. De code die in deze sectie wordt weergegeven, is slechts een deel van de codebases van de applicatie.

Figuur 8.2: Flutter code snipper voor het maken van een notificatielijst.

```dart
import 'package:flutter/material.dart';

class H7 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Chapter 7 Code complexity"),
      ),  // AppBar
      body: Container(
        child: Padding(
          padding: EdgeInsets.only(top: 50.0, bottom: 40),
          child: ListView.builder(
              itemCount: 20,
              itemBuilder: (context, index) {
                return Container(
                  decoration: BoxDecoration(
                      border: Border(
                    bottom: BorderSide(
                        style: BorderStyle.solid, color: Colors.black12),  // BorderSide
                  )),  // Border, BoxDecoration
                  child: ListTile(
                    contentPadding: EdgeInsets.only(left: 15),
                    title: Text("Notification"),
                  ),  // ListTile
                );  // Container
              }),  // ListView.builder
        ),  // Padding
      ),  // Container
    );  // Scaffold
  }
}
```

Tabel 8.1: Ontwikkelingstijd voor elke applicatie

	Totaal
Android	26 Uur
Flutter	20 Uur

Figuur 8.2 laat zien hoe Flutter het toe staat om de visuele opbouw van een scherm in hetzelfde codeblok te verwerken als de functionele code achter het scherm. In de afbeelding is er een child parent relatie voor de widget elementen. De code laat zien dat er een widget is die is gebouwd om geneste widgets te retourneren. Het kind naar de hoofdcontainers Padding, toont het maken van een ListView Flutter-widget. Deze weergave retourneert een lijstweergave met 20 items en 20 containeritems.

In figuur **??** verklaart de Android Kotlin-toepassingscode het weergavemodel en importeert de benodigde items voor de code. Alles gebeurt in de onCreateView-functie, die de XML-lay-out voor de notificatielijst opblaast en een array creëert met dezelfde strings "Notification"die het als gegevens injecteert in het reeds bestaande ListView XML-doel met de id "products". De native code van Android heeft veel omgevingsspecifieke variabelen waarmee een ontwikkelaar rekening moet houden.

Deze code is relatief kort voor zijn doel en is gemakkelijk gestructureerd voor een beginner in mobiele ontwikkeling. Er zijn taalspecifieke delen van de code, maar verder zou het waarschijnlijk kunnen worden begrepen door iemand die kennis heeft van andere talen.

Development time
Zoals te zien valt in tabel 8.1, is het meeste tijd naar de ontwikkeling van de native Android app gegaan. De toegankelijkheid van Dart samen met krachtige simpliciteit van Flutter leidde tot een snelle vooruitgang binnen de ontwikkeling van de onderzoeksapplicatie. De hot reload functionaliteit die het toestaat om een applicatie in enkele seconden up te daten op basis van wijzigingen in de codebase is dan ook een goede bijdrage aan het platform. Android daarentegen verwacht meer boilerplate code met een vaste structuur. De opbouw van een layout is gemakkelijk met de drag en drop functionaliteit, maar het linken van deze views aan code is dan weer moeilijker.

8.3 Conclusie

9. Tooling

Een besturingssysteem is een interface tussen de gebruiker en de hardwarecomponenten. Het voert verschillende taken uit. Een besturingssysteem biedt de gebruiker een Graphical User Interface (GUI) of Command Line Interface (CLI) aan om taken uit te voeren. Sommige besturingssystemen bieden alleen een CLI of GUI aan, terwijl andere zowel GUI als CLI aanbieden. Een GUI bestaat uit bedieningselementen of widgets om met de computer te communiceren. Aan de andere kant moet de gebruiker bij gebruik van de CLI opdrachten invoeren om de taken uit te voeren. Over het algemeen is GUI gebruiksvriendelijker, maar de uitvoeringssnelheid is hoger in CLI.

Een CLI opdracht wordt dus uitgevoerd aan de hand van een commando.

Nu is het mogelijk om een aantal extra commandos te installeren die deel uitmaken van een overkoepelende tool.

Dit hoofdstuk onderzoek de verschillende commandos van de Flutter tool en vergelijkt deze met de verschillende Android tools en hun gelijkende commandos.

9.1 Opzet

Een Software Development Kit (SDK) is een verzameling softwareontwikkelingstools in één installeerbaar pakket. Ze vergemakkelijken het maken van applicaties door een compiler, debugger en misschien een softwareframework te bevatten. Ze zijn normaal gesproken specifiek voor een combinatie van hardwareplatform en besturingssysteem. Een SDK voorziet een set tools, bibliotheken, relevante documentatie, codevoorbeelden, processen en/of handleidingen waarmee ontwikkelaars softwaretoepassingen op een specifiek platform kunnen maken.

9.2 Android

De Android SDK bestaat uit meerdere tools die nodig zijn voor app-ontwikkeling. De tools kunnen geinstalleerd en bijgewerkt worden met SDK Manager van Android Studio of de command line tool sdkmanager. Alle tools worden gedownload naar de Android SDK-map. Aangezien de aard van het onderzoek een vergelijkende studie is werden hier enkel de commandos aangehaald met een Flutter tegengestelde. Het aanhalen van alle commandos voor elke tool zou te veel en niet interessant zijn.

Android SDK Command-Line Tools (android_sdk/cmdline-tools/version/bin/)	
apkanalyzer	Geeft inzicht in de opbouw van de APK nadat het build proces klaar is.
avdmanager	Staat het toe om Android Virtual Devices (AVD's) te maken en beheren vanaf de command line.
lint	Scant de code en helpt bij het identificeren en corrigeren van problemen met de structurele kwaliteit.
sdkmanager	Staat het toe om Android SDK packages te installeren, te updaten en te verwijderen.

Android SDK Build Tools (android_sdk/build-tools/version/)	
Deze package is vereist om Android apps te bouwen. De meeste tools in deze package worden aangeroepen door de build-tools.	
aapt2	Parses, indexes en compiles Android resources in een binair formaat dat is geoptimaliseerd voor het Android-platform en verpakt de gecompileerde resources in een enkele uitvoer.
apksigner	Ondertekend APK's en controleert of de APK handtekeningen met succes zullen worden geverifieerd op alle platform versies die een bepaalde APK ondersteunt.
zipalign	Optimaliseert APK-bestanden door ervoor te zorgen dat alle niet-gecomprimeerde gegevens beginnen met een bepaalde uitlijning ten opzichte van het begin van het bestand.

Android Emulator (android_sdk/emulator/)	
Deze package is vereist om de Android Emulator te gebruiken.	
emulator	Een op QEMU gebaseerde apparaat emulatie tool die kan gebruikt worden om de applicaties te debuggen en te testen in een echte Android-runtime omgeving.

Android SDK Platform Tools (android_sdk/build-tools/version)	
Deze tools worden bijgewerkt voor elke nieuwe versie van het Android-platform om nieuwe functies te ondersteunen (en soms om de tools te verbeteren) en elke update is achterwaarts compatibel met oudere platformversies.	
adb	Android Debug Bridge (adb) is een veelzijdige tool waarmee de status van een emulatori of Android-apparaat kan beheert worden. Het kan ook gebruikt worden om een APK op een apparaat te installeren.
logcat	Is een tool die via adb wordt aangeroepen om app- en systeem logs te bekijken.

9.3 Flutter

De Flutter command-line tool is hoe ontwikkelaars (en IDEs aangestuurd door ontwikkelaars) omgaan met Flutter.

Voor het kunnen werken met Flutter moet dus eerst de Flutter SDK geïnstalleerd worden. Deze SDK bevat onder andere een aantal commandos die kunnen uitgevoerd worden in de terminal. Flutter commandos worden voorafgegaan door het flutter keyword. Zo weet het besturingssysteem dat een commando wordt gebruikt van de flutter tool.

De dart-tool is een command line interface voor de Dart SDK. De tool is beschikbaar, ongeacht hoe de Dart SDK gedownload werd (expliciete downloadt of alleen de Flutter SDK downloadt). Indien het mogelijk is om de flutter tool te gebruiken is dit het betere alternatief.

Hieronder staat een lijst met de mogelijke flutter en dart commandos.

Flutter command-line tool (android_sdk/build-tools/version)	
Deze commando's worden als volgt opgebouwd: flutter <commando> <optie> <parameter>	
analyze	Analyseert de Dart broncode van het project. Gebruik dit in plaats van dartanalyzer.
assemble	Verzamel en maak Flutter resources.
attach	Verbind het toestel met een draaiende applicatie
build	Flutter build commando's
channel	Toont een lijst van mogelijke flutter channels, of verander van channel.
config	Configureer Flutter instellingen. Om een instelling te verwijderen, configureer deze als een lege string.
create	Maakt een nieuw project aan.
devices	Maakt een lijst van alle aangesloten apparaten.
doctor	Toont informatie over de geïnstalleerde tooling.
downgrade	Downgrade Flutter naar de laatste actieve versie voor de huidige channel.
drive	Voert Flutter Driver-tests uit voor het huidige project.
emulators	Lijst, start en maak emulators.
format	Formateert Flutter broncode. Gebruik dit in plaats van dartfmt.
gen-l10n	Genereert localizations voor het Flutter-project.
install	Installeer een Flutter app op een aangesloten apparaat.
logs	Toon log uitvoer voor draaiende Flutter apps.
pub	Werkt met packages.
run	Voert een Flutter applicatie uit
symbolize	Symboliseer een stacktracering van de door AOT gecompileerde Flutter applicatie.
test	Voert tests uit in de huidige package.
upgrade	Upgrade Flutter.

Dart command-line tools (bin/dart)	
Deze commando's worden als volgt opgebouwd: dart <commando> <optie> <parameter>	
analyze	Analyseert de Dart broncode van het project.
compile	Compileert Dart naar verschillende formaten.
create	Maakt een nieuw project
fix	Past geautomatiseerde oplossingen toe op de Dart broncode.
format	Formatteert Dart broncode.
migrate	Ondersteunt migratie naar null-safety.
pub	Werkt met packages.
run	Voert een Dart progamma uit
test	Voert tests uit in de huidige package
(leeg)	Voert een Dart-programma uit; identiek aan de reeds bestaande Dart VM-opdracht.

9.4 Conclusie

Aangezien Flutter geen toegewijde IDE heeft moeten meer commando's uitgevoerd worden in de terminal.

10. Appendix

10.1 Hello world template

De eerste Flutter app gemaakt in het kader van dit onderzoek was een welbekende en gebruikelijke Hello World applicatie. Een Hello World app klinkt alle ontwikkelaars bekend in de oren. Het is namelijk een standaard template die niet meer doet dan het tonen van een Hello World tekst op een scherm. Het is zeer minimaal in aantal lijnen code, wat ergens wel logisch is, en zorgt voor een heel kleine, compacte APK. Om enige vorm van realisme in de ontwikkeling van deze voorbeeld applicaties te integreren, zal er gekozen worden om de builds van deze applicaties via een release schema uit te voeren. Deze manier van builden optimaliseert de code in zijn geheel, zoals de applicatie terug te vinden zou zijn op de Play Store bijvoorbeeld.

10.2 Opstart procedures van een applicatie

Een cold start houdt in dat de applicatie vanaf nul opgestart wordt. Het systeem heeft namelijk, tot nu, nog niks gedaan om de applicatie zijn proces te starten. Cold starts komen voor wanneer het systeem net opgestart is of dat het systeem de applicatie net zelf afgesloten heeft (bijvoorbeeld omdat het systeem geheugen te kort had). Bij de cold start van een applicatie moet het systeem volgende stappen ondernemen:

1. Laden en lanceren van de applicatie
2. Tonen van een blanco start venster voor de applicatie direct na het lanceren ervan
3. Creëren van het applicatie proces

Van zodra het systeem klaar is met het creëren van het hierboven vermeld applicatie proces, is het de verantwoordelijkheid van de applicatie om de volgende stappen in zijn *lifecycle* te ondernemen. Deze *lifecycle* ligt echter buiten de scope van dit onderzoek.

Een hot start is een andere vorm hoe de applicatie opgestart kan worden. Deze vorm is veel eenvoudiger en vergt minder werk als een cold start. Het enige wat het systeem bij een hot start dient te doen is de view naar de voorgrond brengen. Wanneer alle activiteiten van de applicatie nog in het geheugen aanwezig zijn, kan de applicatie enkele stappen in het opstart proces vermijden. Dit heeft als gevolgd dat het opstarten van de applicatie een pak sneller gaat.

Een warm start dan ten slotte, is een combinatie van de cold en hot start. Er zijn verschillende toestanden die als warm start kunnen bekeken worden. Wanneer een gebruiker weg navigeert van de applicatie en vervolgens direct de applicatie opnieuw opstart, kan dit beschouwd worden als een warm start. Het proces kan mogelijks verder blijven lopen, maar de applicatie moet de view opnieuw creëren vanaf nul. Een ander voorbeeld van een warm start is wanneer het systeem de applicatie uit het geheugen verwijdert, maar de gebruiker de applicatie terug opstart. Het proces en de view moeten opnieuw gestart worden, maar het systeem kan enigszins profiteren van de opgeslagen instantiebundel.

10.3 Threads

De dag van vandaag hebben smartphones meerdere processoren voor het uitvoeren van taken of processen. Deze processoren zijn in staat om verschillende taken gelijktijdig uit te voeren. Dit noemt men *multi-processing*.

Om de processoren efficiënter te gebruiken kan het besturingssysteem een applicatie verplichten om meer dan één thread van uitvoering te creëren binnen een proces. Dit noemt men *multi-threading*. Dit kan vergeleken worden met het lezen van meerdere boeken tegelijkertijd en wisselen tussen elk boek achter een hoofdstuk. In dit scenario is de lezer gelijk aan de processor, alle boeken samen zijn gelijk aan het uit te voeren proces, één boek gelijk aan een thread en het lezen van dat boek gelijk aan het uitvoeren van een thread. Het is echter niet mogelijk om in meerdere boeken tegelijkertijd te lezen.

Voor het beheren van deze threads is veel overhead nodig. Zo is er nood aan een *Schedular*. Deze kijkt onder andere naar prioriteit van alle threads maar houdt ook rekening met het uitvoeren en finaliseren van al deze threads. Vervolgens is er de *Dispatcher*, deze houdt zich bezig met het aanmaken van threads. In het voorbeeld kan dit vergeleken worden met een persoon die de boeken die gelezen dienen te worden bezorgd en een context aanbiedt waarin dit moet gebeuren. De context kan vergeleken worden met een speciale lees kamer. Sommige contexts zijn beter voor UI taken, andere beter voor I/O taken.

Ook is interessant om weten dat gebruikers applicaties meestal een *main thread* hebben. Deze wordt uitgevoerd in de voorgrond en kan andere threads dispatchen in de achtergrond.

11. Conclusie

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

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

De alom bekende smartphones zijn de dag van vandaag niet meer weg te denken en hoewel de markt verzadigd is, kunnen verschillende gelijkenissen getrokken worden tussen de toestellen. Allereerst maakt elk toestel gebruik van een besturingssysteem; voor Apple is dit iOS, terwijl Samsung het Android besturingssysteem gebruikt. Een ander aspect dat ze delen, is het gebruik van mobiele apps. Alvorens een ontwikkelaar begint met het schrijven van een app zal hij eerst moeten kiezen op welk(e) besturingssysteem/besturingssystemen deze app zal draaien. Uit deze keuze komen twee mogelijkheden voort.

Enerzijds native app ontwikkeling, dit zijn apps die rechtstreeks ontwikkeld worden voor een bepaald besturingssysteem. Anderzijds cross-platform app ontwikkeling, dit zijn apps die ontwikkeld en vervolgens gecompileerd kunnen worden voor meerdere besturingssystemen. Een aantal jaar geleden was dit een voor de hand liggende keuze, aangezien cross-platform ontwikkeling nog in zijn jonge schoentjes stond. Tegenwoordig wordt het aanzien als een kost besparend alternatief door het hergebruik van resources. Onder cross-platform ontwikkeling vinden we verschillende frameworks terug die kunnen gebruikt worden. De grootste zijn: Flutter, Xamarin, React Native en Ionic. Dit onderzoek zal zich toespitsen op Flutter, een jong ontwikkelingsplatform waar momenteel relatief weinig onderzoek naar verricht werd. Desondanks heeft het platform veel potentieel en

een snelgroeiende gebruikersbasis. Dit onderzoek zal een beeld schetsen van de voor- en nadelen van beide platformen. Het softwareontwikkelingsbedrijf NextApps, wil met behulp van de resultaten van dit onderzoek een beter inzicht krijgen in de werking van cross-platform development met het Flutter framework. Dit onderzoek zal een antwoord proberen vormen op de volgende onderzoeksvragen: Wat zijn de voor- en nadelen van app ontwikkeling in Flutter in vergelijking met native Android?, Is Flutter al matuur genoeg om te aanschouwen als volwaardig alternatief op native app ontwikkeling?, Is Flutter toegankelijker voor nieuwe ontwikkelaars?.

A.2 State-of-the-art

Flutter, ontwikkeld door Google, is een relatief jong framework waarin, volgens de site[1], mooie, native gecompileerde mobiele-, web- en desktopapplicaties kunnen ontwikkeld worden. Het platform werd aangekondigd door Google in 2015, maar was pas echt gangbaar in december 2018 toen de eerste stabiele versie uitkwam. Flutter is opensource, wat wil zeggen dat er vrije toegang is tot de bronmaterialen. Dit zorgt voor een verdere ontwikkeling van het framework en bevordert een hoog niveau van innovatie. De jonge aard van Flutter betekent anderzijds dat nog niet veel research over het framework verschenen is. De meeste papers en artikels vergelijken de verschillende cross-platform frameworks onderling om zo een beeld te schetsen van alle voor- en nadelen van elk platform. Dit helpt bij het kiezen van een cross-platform framework maar beantwoordt niet de vraag: zou het beter zou zijn om een app native te ontwikkelen?

Het onderzoek zal gebruik maken van drie recente papers die soortgelijke onderzoeksvragen hadden. De eerst paper, van onderzoeker Bracke (2020); vertoont verscheidene gelijkenissen met de onderzoeksvragen van deze paper en onderzocht Flutter tegenover native Android ontwikkeling. De tweede paper van Olsson (2020); onderzoekt het Flutter framework tegenover native app ontwikkeling, wat betekend dat het onderzoek ook rekening houdt met iOS, wat deze paper niet zal doen. De laatste paper van Cheon en Chavez (2020); is een paper over de creatie van een Flutter app uit een reeds bestaande Android app. Voor de ontwikkeling van de Flutter app, zal de gids van Payne (2019) gebruik worden. Verder biedt de Flutter site[2] duidelijke en overzichtelijke documentatie aan.

A.3 Methodologie

De uitvoering van het experiment van dit onderzoek omvat het schrijven van twee applicaties. De eerste app zal geschreven worden in het Flutter framework, gebruikmakend van de Dart programmeertaal. De andere app zal geschreven worden in native Android,

[1] https://flutter.dev
[2] https://flutter.dev/docs

gebruikmakend van Kotlin[3]. Als onderdeel van het uit te voeren experiment, zal het schrijven van beide applicaties gedocumenteerd worden. Het onderzoek zal een finale conclusie vormen op de onderzoeksvragen, gebaseerd op de analyse van de onderzoeksresultaten. Op basis van deze resultaten kan het Flutter platform worden beoordeeld. Ten slotte zal het onderzoek de voor- en nadelen van elk systeem oplijsten.

Beide apps zullen ontwikkeld worden met de Android SDK. De twee apps zullen ook op een gelijkaardige manier ontworpen worden om de user experience tussen de twee apps zo gelijk mogelijk te houden. Als maatstaaf voor de ontwikkelde applicaties worden een aantal minimumvereiste opgelegd zoals een goede performantie en een mooie visuele samenhang. Bij het schrijven van de applicaties zullen verscheidene aspecten bekeken en vergeleken worden. Het onderzoek zal gebruik maken van een lijst met richtlijnen. Deze zullen vermelden wat de vooropgestelde functionaliteiten van de apps zullen zijn.

De te vergelijken aspecten:

- Grootte van de uitvoeringsbestanden en opstartsnelheid van de app
- CPU gebruik van de app
- Gebruik van online APIs
- Security
- Beschikbare libraries en code complexiteit
- Creatie van views
- Asynchroon werken
- Beschikbare tools

A.4 Verwachte Resultaten

Uit eerdere onderzoeken blijkt Native app ontwikkeling een beter alternatief te bieden op vlak van performantie en opstartsnelheid. Een Flutter APK moet volgende zaken bevatten: de core engine, Java-, app- en framework code, het licentie bestand en ICU data. Een Flutter APK heeft een minimale grote van 4.7MB omdat het zijn eigen resources nodig heeft. Ook zal rekening gehouden worden met de leeftijd van het Flutter platform, hier worden nog wat imperfecties verwacht. Door het grote potentieel en de vele steun die Flutter heeft, zal dit naar de toekomst toe, hoogstwaarschijnlijk worden opgelost. Van Flutter wordt verwacht dat het toegankelijker is voor nieuwe applicatieontwikkelaars. Ook wordt verwacht dat Flutter de bovenhand zal nemen op vlak van code complexiteit en creatie van views.

[3]https://kotlinlang.org

A.5 Verwachte conclusies

Op basis van de resultaten van het onderzoek zal een conclusie op de onderzoeksvragen gevormd worden. In essentie zal deze conclusie antwoorden op de vraag, is Flutter al ver genoeg ontwikkeld om aanschouwd te worden als volwaardige tegenhanger van native development? Gebruikmakend van de lijst met richtlijnen en de behaalde scores, zal hier hopelijk een duidelijk antwoord gegeven worden.

Bibliografie

Bracke, N. (2020). *Android Native Development in Kotlin versus het Flutter Framework, een vergelijkende studie*. https://scriptie.hogent.be/2019-2020/322_201639090_PBA-TIN_scriptie.pdf

Cheon, Y. & Chavez, C. (2020). *Creating Flutter Apps from Native Android Apps* (masterscriptie). University of Texas at El Paso. https://scholarworks.utep.edu/cgi/viewcontent.cgi?article=2485&context=cs_techrep

Dang, D. & Skelton, D. (2019). Teaching mobile app development: choosing the best development tools in practical labs: Proceedings of the 10th Annual Conference of Computing and Information Technology Education and Research in New Zealand (D. E. Erturk, Red.). *ITx New Zealand's Conference of IT*, 26–31. https://www.researchgate.net/profile/Hanif_Deylami/publication/344217702_Recent_educational_developments_in_Cyber_Security/links/5f5d5404299bf1d43cff9236/Recent-educational-developments-in-Cyber-Security.pdf#page=28

Faust, S. (2020, januari 31). *Using Google s Flutter Framework for the Development of a Large-ScaleReference Application* (masterscriptie). Technical University Cologne. https://epb.bibl.th-koeln.de/frontdoor/deliver/index/docId/1498/file/flutter-for-the-dev-of-large-scale-ref-app.pdf

Fayzullaev, J. (2018). *Native-like Cross-PlatformMobile Development* (masterscriptie). University of Applied Sciences Xamk South-Eastern Finland. https://www.theseus.fi/bitstream/handle/10024/148975/thesis_Jakhongir_Fayzullaev.pdf?sequence=1&isAllowed=y

Flutter. (2021, maart 4). *Flutter Engage 2021*. Flutter. https://www.youtube.com/watch?v=yll3SNXvQCw

Latture, A. D. (2020, april 11). *Backdrop: An Exploration of Flutter* (onderzoeksrap.). Grand Valley State University. https://scholarworks.gvsu.edu/cgi/viewcontent.cgi?article=1346&context=cistechlib

Olsson, M. (2020, juni 13). *A Comparison of Performance and Looks Between Flutter and Native Applications: When to prefer Flutter over native in mobile application development*. https://www.diva-portal.org/smash/get/diva2:1442804/FULLTEXT01.pdf

Payne, R. (2019). *Beginning App Development with Flutter: Create Cross-Platform Mobile Apps*. Apress.