CHALMERS

GÖTEBORGS UNIVERSITET

Att välja cross-platform-ramverk för mobilapplikationsutveckling

En jämförelse av Xamarin, Xamarin.Forms och Flutter

Examensarbete inom högskoleingenjörsprogrammet Datateknik

DANIEL DUVANÅ
DAVID SVENSSON

Institutionen för Data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Att välja cross-platform-ramverk för mobilapplikationsutveckling

En jämförelse av Xamarin, Xamarin.Forms och Flutter

DANIEL DUVANÅ
DAVID SVENSSON

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2019

Att välja cross-platform-ramverk för mobilapplikationsutveckling
En jämförelse av Xamarin, Xamarin.Forms och Flutter

DANIEL DUVANÅ
DAVID SVENSSON

Examinator: Jonas Almström Duregård

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
SE-412 96 Göteborg
Sverige

Göteborg, Sverige 2019

Abstract

The purpose of the project is to evaluate differences in performance between the cross-platform frameworks Xamarin (consisting of Xamarin.Android and Xamarin.IOS), Xamarin.Forms and Flutter. Another part of the purpose is to evaluate if it's technically possible, for a specific company and their application, to switch from Xamarin.Forms to Flutter, and if so; what advantages and disadvantages such a switch would bring.

A set of small applications was developed for each framework in order to test the performance of the frameworks with regard to startup time and installation size. In addition to this, a prototype application was developed to investigate if Flutter is suitable for applications where there's a technical requirement for Bluetooth communication.

No significant differences in performance between the frameworks were observed on IOS. On Android, the test results show that Flutter performs significantly better in all tests compared to Xamarin.Forms, and have slightly better performance than Xamarin.Android. Flutter also provides useful tools that can help improve the development process over Xamarin and Xamarin.Forms. The project also resulted in the development of a working prototype application, meeting the technical requirements. For developing new cross-platform applications from the start, we recommend using Flutter. However, in the case of the company, it might be possible to significantly improve performance of an existing application (on Android) by switching from Xamarin.Forms to Xamarin without having to rework their entire codebase.

This thesis is written in Swedish.

Keywords: cross-platform, mobile applications, applications, application development, bluetooth, Xamarin, Xamarin.Forms, Flutter.

Sammanfattning

Arbetets syfte är att utvärdera skillnader i prestanda mellan cross-platformramverken Xamarin (bestående av Xamarin.Android och Xamarin.IOS), Xamarin.Forms, och Flutter. I arbetets syfte ingår också att utvärdera om det är tekniskt möjligt att, för ett företag och deras specifika applikation, byta ramverk från Xamarin.Forms till Flutter samt vilka för-/nackdelar ett sådant byte skulle kunna medföra.

För att utvärdera ramverkens prestanda har en uppsättning testapplikationer tagits fram för varje ramverk. M.h.a. dessa applikationer har sedan skillnader i uppstartstid och installationsstorlek uppmätts. Förutom prestandajämförelserna mellan ramverken har också en prototypapplikation tagits fram för att undersöka om Flutter som ramverk lämpar sig för en speciell typ av applikation där det finns krav på bl.a. Bluetooth-kommunikation.

Inga betydande prestandaskillnader observerades mellan ramverken på IOS. På Android är Flutter det ramverk som presterar bäst, tätt följt av Xamarin.Android och båda dessa presterar betydligt bättre än Xamarin.Forms. Dessutom erbjuder Flutter fördelaktiga verktyg som underlättar utvecklingsarbetet. Arbetet resulterade också i en fungerade prototypapplikation som uppfyller de krav som ställts. Om en applikation utvecklas från grunden rekommenderar vi Flutter som ramverk. I företagets specifika fall kan eventuellt en stor prestandaförbättring uppnås (på Android) genom att byta från Xamarin.Forms till Xamarin, utan att byta ut en stor del av kodbasen.

Nyckelord: cross-platform, mobilapplikationer, applikationer, applikationsutveckling, bluetooth, Xamarin, Xamarin.Forms, Flutter.

Förord

Vi skulle vilja tacka företaget som erbjudit oss möjligheten att utföra vårt examensarbete på plats på deras kontor. Ett stort tack till vår eminenta handledare och alla de utvecklare på företaget som hjälpt oss under utvecklingen av prototypapplikationen.

Ett stort tack också till vår handledare på Chalmers, Uno Holmer, som stöttat och hjälpt oss genom hela projektet.

Företaget som arbetet utförts hos önskar förbli anonyma i denna rapport och kommer därför i resten av rapporten att omnämnas som ”företaget”.

Förkortningar och begrepp

Android – Operativsystem för smartphones.

Android Studio – IDE som främst används vid Android-utveckling i en rad olika språk.

AOT – Ahead of Time, kompileringsstrategi där bytekod kompileras och sparas mellan körningar av applikationen.

APK-fil – Android Package, applikationsfil för Android-plattformen.

Arduino – Utvecklingshårdvara och experimentplattform.

Binär storlek – En applikationsfils storlek, alltså applikationens storlek innan installation.

BLE – Bluetooth Low Energy, vidareutveckling av Bluetooth-standarden som inkluderas fr.om. Bluetooth 4.0.

Bytekod – Kod ej är bunden till specifik hårdvara, avsedd att tolkas av en VM eller kompileras till maskinkod.

Cache – Data som mellanlagras för att snabbt vara tillgänglig. Används ofta för att snabba upp exekvering av ett tidigare exekverat program.

Dart - Objektorienterat programmeringsspråk.

Flutter – Cross-platform-ramverk utvecklat av Google.

IDE – Integrated Development Environment, utvecklingsmiljö avsedd att underlätta programmerarens arbete. Innehåller oftast kompilator, felsökningsprogram och textredigerare.

Installationsstorlek – Storleken av en applikation efter installation på enhet.

Ionic – Cross-platform-ramverk utvecklat av Ionic.

IOS – Operativsystem för smartphones.

IoT – Internet of Things.

Java – Objektorienterat programmeringsspråk.

JavaScript – Skriptprogramspråk främst använt i webbteknologier.

JavaScriptCore – JavaScript-motor som tolkar JavaScript-kod på IOS-plattformen.

JIT – Just in Time, kompileringsstrategi där bytekod kompileras till maskinkod precis innan den exekveras.

JSON – JavaScript Object Notation, ett textbaserat filformat ofta använt för att skicka data över webben.

JVM – Java Virtual Machine, virtuell maskin speciellt avsedd för att exekvera Java bytekod.

Kotlin – Objektorienterat programmeringsspråk.

LLVM – Low Level Virtual Machine, programbibliotek som används för att optimera programkod för att i sin tur möjliggöra mer effektiv maskinkod.

Mesh-nätverk – Internt Bluetooth-nätverk bestående av två eller flera enheter (noder).

Native-applikation – Applikation utvecklad att endast köras på en plattform, antingen IOS eller Android.

Objective-C – Objektorienterat programmeringsspråk, bygger vidare på språket C.

ProGuard – Verktyg som används för att försvåra demontering av Android-applikationer.

React Native – Cross-platform-ramverk utvecklat av Facebook.

SDK – Software Development Kit, en uppsättning verktyg för utveckling av applikationer riktade mot en specifik platform.

Smart belysning – Appstyrd belysning, möjlighet finns att styra belysningen via en applikation på smartphone.

Swift – Multi-paradigmatiskt programspråk.

UI – User Interface, användargränssnitt i t.ex. en mobilapplikation.

VM – Virtual Machine, virtuell maskin som möjliggör att kod kan exekveras på samma sätt oavsett den underliggande plattformen som den virtuella maskinen körs på.

Xamarin – Cross-platform-ramverk utvecklat av Microsoft.

Xamarin.Forms – Cross-platform-ramverk utvecklat av Microsoft

Innehållsförteckning

1. Inledning

1.1 Bakgrund

Antalet smartphoneanvändare över hela världen har estimerats till 3 miljarder år 2018 och förväntas växa under kommande år [1]. Som en följd av detta är också marknaden för applikationer till smartphones väldigt stor, med över 175 miljarder installationer under 2017 på de två dominerande operativsystemen IOS och Android [2].

Mer än 99% av alla sålda smartphones de senaste åren hade antingen iOS (ca. 15%) eller Android (ca. 85%) som operativsystem [3]. Om utvecklare vill göra sina applikationer tillgängliga för nästan alla smartphoneanvändare behövs alltså att applikationerna utvecklas till både IOS och Android. Traditionellt sett innebär detta att utveckla två versioner av samma applikation i olika programmeringsspråk; Kotlin eller Java för Android och Swift eller Objective-C för IOS. Resultatet blir två separata, icke-kompatibla, kodbaser som måste byggas och underhållas för en och samma applikation. Detta kan leda till flera olika problem, såsom högre utvecklingskostnader och osymmetrisk funktionalitet och användarupplevelse mellan IOS- och Androidversionen.

En potentiell lösning på dessa problem är plattformsoberoende utveckling med så kallad *cross-platform*-mjukvara. Det innebär teknologi som möjliggör utveckling av mjukvara med *en* kodbas, som sedan kan köras på flera plattformar utan att behöva skrivas om till varje enskild plattform [4]. Denna rapport kommer enbart behandla cross-platform-teknologi som möjliggör simultan utveckling av applikationer för IOS och Android. I dagsläget finns flera cross-platform-lösningar tillgängliga, såsom Xamarin och Xamarin.Forms av Microsoft, React Native av Facebook och Flutter av Google.

Arbetet har utförts tillsammans med ett företag som tillverkar och säljer system för smart belysning som kan styras via deras smartphoneapplikation. Företagets nuvarande applikation är byggd med Xamarin.Forms. Detta medför fördelar såsom lägre utvecklingskostnad och högre paritet i funktionalitet mellan IOS- och Androidversionen, men också nackdelar; framförallt har cross-platform-lösningar nästan alltid sämre prestanda än *native*-utveckling för IOS och Android [5]. Detta är också vad företaget är missnöjda med, särskilt hur lång tid det tar starta deras

applikation, både på Android och IOS. Deras användare öppnar applikationen många gånger per dag för att styra sin belysning, men spenderar bara en kort stund i applikationen varje gång. Detta gör att en långsam uppstartstid snabbt kan bli ett betydande problem.

1.2 Syfte

Arbetets syfte är att utvärdera skillnader i grundläggande prestanda mellan cross-platform-teknologierna Xamarin.Forms, Xamarin och Flutter. I syftet ingår också att utvärdera om det är tekniskt möjligt för företaget att byta ramverk från Xamarin.Forms till Flutter samt vilka för-/nackdelar ett sådant byte skulle medföra.

1.3 Frågeställningar

- Hur skiljer sig uppstartstid och installationsstorlek mellan minimala applikationer skapade med Xamarin.Forms jämfört med motsvarande applikationer skapade med Xamarin och Flutter?

- Vilka för-/nackdelar skulle Flutter ha jämfört med Xamarin.Forms för företaget om de väljer att byta ramverk, t.ex. med hänsyn till utvecklingsprocessen?

- Kan nödvändig funktionalitet uppnås i Flutter för den typ av applikationer som företaget utvecklar?

1.4 Avgränsningar

Med hänseende till examensarbetets tidsbegränsning så har följande avgränsningar gjorts.

- Prestandamätningar kommer enbart utföras på ramverken Xamarin.Forms, Xamarin och Flutter. Xamarin.Forms inkluderades då det är det ramverk som företaget använder i dagsläget. Xamarin inkluderades för att undersöka om eventuella prestandavinster kan motivera ett byte av ramverk från Xamarin.Forms för företaget. Flutter inkluderades för att det i skrivande stund är det nyaste ramverket på marknaden, vilket gör att mycket få tillförlitliga prestandamätningar existerar för just Flutter.
- De applikationer som utvecklas med Xamarin.Forms, Xamarin och Flutter kommer enbart testas på tre olika enheter; en iPhone och två Android smartphones.
- Företagets hårdvaruprodukter kommer inte att modifieras på något sätt.

1.5 Läsarguide

Kapitel 2 förklarar vad ett cross-platform-ramverk är för något och dess användningsområden, behandlar grundläggande kompileringstekniker för mobila applikationer, ger en grundläggande förståelse för hur Bluetooth fungerar samt förklarar vad för typ av system företaget erbjuder sina kunder i dagsläget. Dessutom ges en översikt över några av de mest populära mobila cross-platform-ramverken som finns på marknaden idag och vad som skiljer dessa åt.

Kapitel 3 förklarar de olika delarna av projektet. Här ges information om både hur testerna mellan ramverken utförts samt tanken bakom prototypapplikationen.

Kapitel 4 berör prototypapplikationens utvecklingsprocess. Här ges detaljerad information om hur applikationens olika beståndsdelar fungerar.

Kapitel 5 redovisar resultaten från jämförelserna mellan ramverken. I detta kapitel ryms resultaten från både Android- och IOS-testerna.

Kapitel 6 visar upp slutresultatet av prototypapplikationens utveckling och ger en insikt i vilka krav från kravspecifikationen som gick att uppfylla.

I kapitel 7 diskuteras de övergripande testresultaten från prestandajämförelserna mellan ramverken och även den metod som använts för att få fram resultaten. Kapitlet berör också prototypapplikationens utvecklingsprocess och de problem som uppstått under projektet.

Kapitel 8 tar upp de samhälls- och miljöaspekter som cross-platform-utveckling har möjlighet att påverka.

Kapitel 9 slutsats för projektet där rapportens frågeställningar besvaras.

2. Teknisk bakgrund

Målet med cross-platform-utveckling är att möjliggöra att en och samma applikation kan köras på flera olika hårdvaru- och/eller mjukvaruplattformar med en enda kodbas, istället för att behöva separata (och ej interoperabla) kodbaser för varje plattform. Det kan t.ex. innebära att en applikation programmeras för både IOS och Android i samma språk och inom samma ramverk. Att ha en enda kodbas istället för två (eller fler) ger flera fördelar såsom att utveckling och underhåll är enklare och högre paritet i funktionalitet mellan plattformarna. Vilket är positivt för såväl utvecklaren som slutanvändaren.

För många företag som väljer cross-platform-utveckling är de potentiella kostnadsbesparingarna det avgörande argumentet. Under ideala förhållanden sker all utveckling i samma programmeringsspråk och allt kodas en gång, för både IOS och Android. Förhoppningen är att detta skall innebära att man kan uppnå ett slutresultat likvärdigt med native-utveckling, fast med färre utvecklare och/eller på kortare tid.

Dock medför cross-platform ramverk inte bara fördelar; ett av de största problemen med ramverken som finns tillgängliga idag är att prestandan är sämre jämfört med native utveckling. T.ex. är uppstartstid, RAM-användning, UI-prestanda samt installationsstorlek på native-applikationer generellt bättre än motsvarande applikationer byggda med cross-platform ramverk. Detta är faktorer som påverkar både utvecklare och slutanvändare.

2.1 Kompileringsstrategier och verktyg

Det finns ett antal olika kompileringsstrategier som kommer att beröras i denna rapport. Huvudsakligen handlar det om Ahead of Time- (AOT) och Just In Time-kompilering (JIT). Dessa två strategier påverkar olika faktorer såsom uppstartstid, binär storlek samt installationsstorlek hos applikationerna. Det finns även verktyg som Proguard och Low Level Virtual Machine (LLVM) som är byggda för att utföra optimeringar på programkoden. Ovan nämnda verktyg och kompileringsstrategier går att kombinera på olika sätt beroende på vilket ramverk och språk som applikationerna utvecklas i.

AOT-kompilering bygger på principen att applikationens kod kompileras till maskinkod som kan exekveras direkt på den underliggande plattformen [6]. Efter kompilering av maskinkoden sparas den undan för att återanvändas vid nästkommande exekveringar [7]. Detta resulterar generellt i snabbare uppstartstid av applikationen men på bekostnad av att både den binära storleken och installationsstorleken ökar.

JIT-kompilering använder sig av ett annat tillvägagångssätt där källkoden först kompileras till bytekod som sparas i applikationen. Vid varje exekvering av applikationen kompileras den nödvändiga bytekoden till maskinkod precis innan exekvering [8]. Efter att applikationen stängs kasseras maskinkoden som kompilerats för exekveringen. Detta medför att applikationernas binära storlek och installationsstorlek minskar men på bekostnad av att uppstartstiden blir längre då maskinkoden måste kompileras på nytt vid varje uppstart.

LLVM är ett bibliotek som används för att optimera och generera maskinkod som sedan kan köras direkt på underliggande plattform. Kod skriven i exempelvis C#, översätts till LLVM:s eget "övergångsspråk" (Intermediate Representation) vilket är det språk som används av LLVM:s kompilator. Kompilatorn har sedan möjlighet att utföra analys och optimeringar av övergångskoden innan maskinkod för målplattformen genereras [9].

Proguard är ett verktyg som används för att optimera Java bytekod [10], vilket hjälper till att minska Android-applikationers storlek genom att ta bort onödiga instruktioner, klasser och metoder i applikationen som inte kommer att användas vid exekvering [11]. Förutom att ta bort onödig kod kan Proguard också göra det svårare att demontera applikationer genom att dölja inkluderade paket, klasser och metoders faktiska namn [10].

2.2 Bluetooth Low Energy

Företagets produkter använder sig av Bluetooth Low Energy för att kommunicera både med varandra och samtidigt med andra enheter som mobiltelefoner. Därför har Bluetooth-kommunikation en central roll i prototypapplikationen och är en nödvändighet för att kunna styra företagets enheter. I tabell 1 återfinns en jämförelse av några nyckelspecifikationer mellan Bluetooth Classic och Bluetooth Low Energy. Tabellen är en omarbetad version av den specifikation för Bluetooth som Bluetooths intresseorganisation, Bluetooth SIG, tagit fram [12].

Tabell 1 - Specifikationer för Bluetooth Classic respektive Bluetooth Low Energy.

Specifikation	**Bluetooth Classic**	**Bluetooth Low Energy**
Maximal bandbredd (teoretisk)	3 Mbit/s	2 Mbit/s
Energiförbrukning	1 (referensvärde)	0,01x-0,5x av referensvärdet
Antal kanaler	79	40
Kanalbredd	1 MHz	2 MHz
Stöd för mesh-nätverk	Nej	Ja
Stöd för strömning av ljuddata	Ja	Nej

Bluetooth Low Energy är en del av Bluetooth 4.0-standarden och är även en del i senare standarder [13, s.1]. Det finns skillnader i hur Bluetooth Low Energy fungerar och vilka applikationer det lämpar sig för jämfört med Bluetooth Classic. Bluetooth Classic lämpar sig framförallt för applikationer som kontinuerligt strömmar data och som har krav att kunna skicka och motta data med en viss bandbredd [14, s.2]. Vanliga applikationer där Bluetooth Classic används är således strömning av musik eller röstsamtal. Bluetooth Low Energy lämpar sig istället för applikationer där kontinuerlig strömning av data inte är ett krav och kommunikation mellan enheter bara sker periodiskt i kortare intervall [12] [15].

Bluetooth Low Energy, har som namnet antyder, fokus på låg energiförbrukning. BLE tillåter enheter att spara ström genom att låta enhetens radio gå ner i standby-läge eller helt stängas av i perioder när den inte används [14, s.4] [13, s.8]. Detta gör att Bluetooth Low Energy lämpar sig väl för applikationer såsom *IoT*-enheter, speciellt batteridrivna sådana då energiförbrukningen kan hållas tillräckligt låg för att kunna driva en enhet i flera år med endast ett knappcellsbatteri [15].

En annan punkt där Bluetooth Low Energy och Bluetooth Classic skiljer sig är stödet för olika nätverkstopologier. Bluetooth Low Energy har stöd för *mesh*-nätverk där en BLE-enhet har möjlighet att kommunicera med andra BLE-enheter samtidigt som den kommunicerar med t.ex. en telefon [16]. Detta kan ge stora fördelar i applikationer där Bluetooths begränsade räckvidd inte räcker till eller där finns ett behov av att Bluetooth-enheterna kan kommunicera direkt med varandra [15].

Bluetooth använder sig av 2,4 GHz-bandet, vilket är ett öppet frekvensspektrum som primärt används av WIFI-nätverk. Bluetooth Classic har 79 st kanaler med 1 MHz kanalbredd tillgängliga och Bluetooth Classic har 40 st kanaler med 2 MHz kanalbredd tillgängliga (se tabell 1) [12]. För att begränsa störningar från trådlösa nätverk och andra enheter som använder sig

av samma frekvensspektrum använder sig Bluetooth av ”adaptive frequency hopping” (AHF), vilket möjliggör att Bluetooth-enheterna sömlöst kan byta frekvens om det finns för mycket störningar på den aktuella frekvensen [17]. Detta minimerar i sin tur den andel paket som behöver återsändas p.g.a. störningar, vilket är fördelaktigt från ett strömförbrukningsperspektiv.

Bluetooth Low Energy-enheter kan primärt kategoriseras som antingen en huvudenhet (central) eller kringutrustning (peripheral) [13, s.10]. Alla Bluetooth-enheter har en GAP (General Access Profile) vilken definierar enhetens roll (huvudenhet/kringutrustning), hur eventuell annonsering sker mot andra enheter och hur anslutningar mellan enheter hanteras [13, ss. 35-37]. Kringutrustningen annonserar ut till andra enheter när den är redo att anslutas till och huvudenheter som finns i närheten kan då ansluta till den.

När anslutningen fullbordats mellan kringutrustning och huvudenhet dikterar kringutrustningens GATT (Generic Attribute Profile) hur kommunikationen, mer specifikt, hur data hanteras och utbyts mellan enheterna [13, ss. 51-52]. GATT innehåller en eller flera services där varje service finns till för att på ett logiskt sätt kategorisera och dela upp den typ av data som ska skickas och/eller mottas av en enhet.

En service innehåller i sin tur ofta en eller flera characteristics. En characteristic är ansvarig för en specifik del data som ska skickas eller mottas, t.ex. ett värde utläst från en sensor [13, ss. 59-61]. Om datan i en characteristic förväntas uppdateras kan en prenumeration på samma characteristic upprättas för att kontinuerligt utläsa data från den.

2.3 Företagets system

I huvudsak utvecklar företaget enheter som styr belysning, dessa enheter installeras på samma sätt som en dimmer eller lampknapp men ger möjligheten att också styra lamporna via en applikation installerad på en Android- eller IOS-enhet. Alla företagets produkter använder sig av Bluetooth Low Energy för att ansluta mot mobila enheter.

Företagets Bluetooth-enheter konfigureras så att de tillhör en plats, såsom ett hus och ett specifikt rum i huset. Alla enheter som konfigurerats för samma plats ingår också i ett mesh-nätverk via Bluetooth, vilket gör att enheterna är uppkopplade mot varandra via Bluetooth samtidigt som telefoner kan ansluta till dem. Mesh-nätverket möjliggör att en telefon kan ansluta mot en enhet i mesh-nätverket via Bluetooth och samtidigt skicka kommandon till en annan enhet, utan att användaren behöver tänka på vilken specifik enhet den är ansluten till. Användaren behöver inte heller befinna sig i närheten av enheten den vill styra, så länge som enheten som styr lampan är inom räckvidd för en annan enhet i mesh-nätverket. Mesh-nätverket ser även till att uppdatera alla enheter om t.ex. en annan enhet har uppdaterat ljusstyrkan på en lampa. På så sätt uppdateras också statusen för de enheter som inte aktivt ändrar ljusstyrkan för en lampa.

2.4 Översikt av ramverk

Det finns ett stort antal ramverk att välja mellan för att utveckla applikationer till IOS och Android med en gemensam kodbas, med olika tekniska lösningar och utvecklingsverktyg. I detta avsnitt ges en översikt av några av de mest populära ramverken för att belysa deras skillnader och likheter. En sammanfattning av ramverkens skillnader och likheter kan ses i tabell 2.

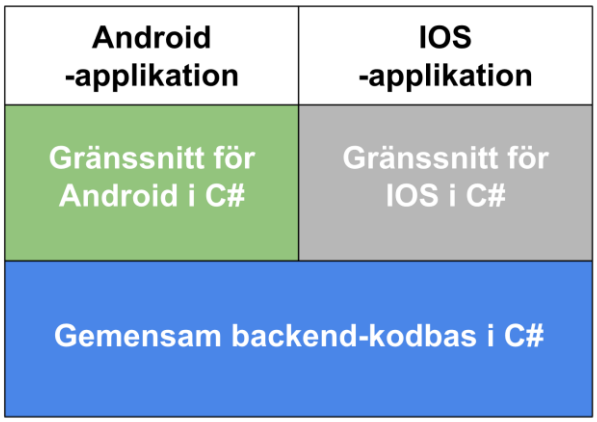
Tabell 2 - Översikt över utvecklingsramverk för IOS och Android.

	Xamarin	**Xamarin.Forms**	**Flutter**	**React Native**	**Ionic**	**IOS SDK**	**Android SDK**
Plattformar	IOS & Android	IOS & Android	IOS & Android	IOS & Android	IOS & Android	IOS	Android
Utvecklare	Microsoft	Microsoft	Google	Facebook	Ionic	Apple	Google
Språk	C#	C#	Dart	JavaScript	JavaScript, HTML, CSS	Swift, Objective-C, C/C++	Java, Kotlin, C/C++
Kompilering IOS	AOT	AOT	AOT	Tolkat med JavaScriptCore	JIT som standard, möjlighet att tolkas med JavaScriptCore	AOT	-
Kompilering Android	JIT, AOT	JIT, AOT	AOT	Kombinerat JIT och tolkat med JavaScriptCore	JIT	-	JIT, AOT
Använder plattformens inbyggda UI-komponenter	Ja	Ja	Nej	Ja	Nej	Ja	Ja
Snabb uppdatering av kod utan fullständig kompilering	Nej	Nej	Ja	Ja	Ja	Nej	Ja
Open source	Ja	Ja	Ja	Ja	Ja	Nej	Ja

2.4.1 Xamarin

Några av de största och mest använda cross-plattform-ramverken är Xamarin.IOS (f.d. MonoTouch) som släpptes 2009 [18] och Xamarin.Android (f.d. Mono for Android) som släpptes 2011 [19]. Båda utvecklade av Microsoft-ägda Xamarin för att underlätta utveckling till flera mjukvaruplattformar genom ett och samma ramverk [20].

Utvecklare kan använda Xamarin för att skriva applikationer till både Android och IOS med en gemensam kodbas för *backend* skriven i C# [21]. Den gemensamma kodbasen kombineras med platformsspecifik kod för gränssnitt (se figur 1), också skriven i C#, som knyts till respektive operativsystems egna komponenter för gränssnitt, vilket resulterar i att applikationerna följer operativsystemets designmönster och övriga utseende.



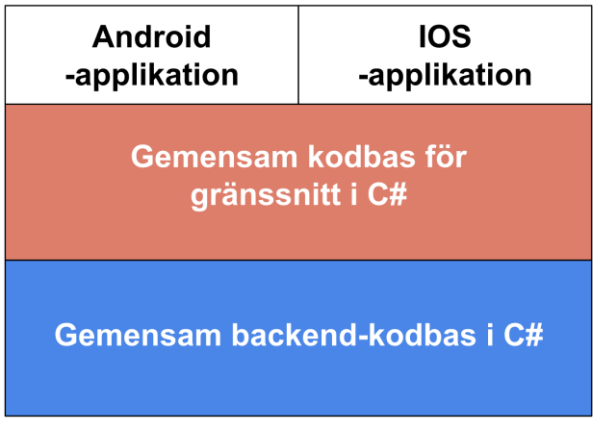
Figur 1 - Arkitektur för Xamarin.Android och Xamarin.IOS.

Applikationer skrivna i Xamarin.IOS/Xamarin.Android körs alla inuti *Mono*:s exekveringsmiljö men med vissa plattformsspecifika skillnader [22] [23]. IOS kör Mono:s exekveringsmiljö tillsammans med IOS egen exekveringsmiljö *Objective-C Runtime*. På grund av begränsningar i IOS använder sig Xamarin.IOS enbart av AOT-kompilering av C#-källkoden, vilken kompileras till maskinkod [22].

I Xamarin.Android körs Mono:s exekveringsmiljö tillsammans med Androids egen exekveringsmiljö *ART* [23]. Applikationer skrivna i Xamarin.Android är som standard JIT-kompilerade, men kan AOT-kompileras för att få bättre prestanda med hänsyn till t.ex. uppstartstid [24].

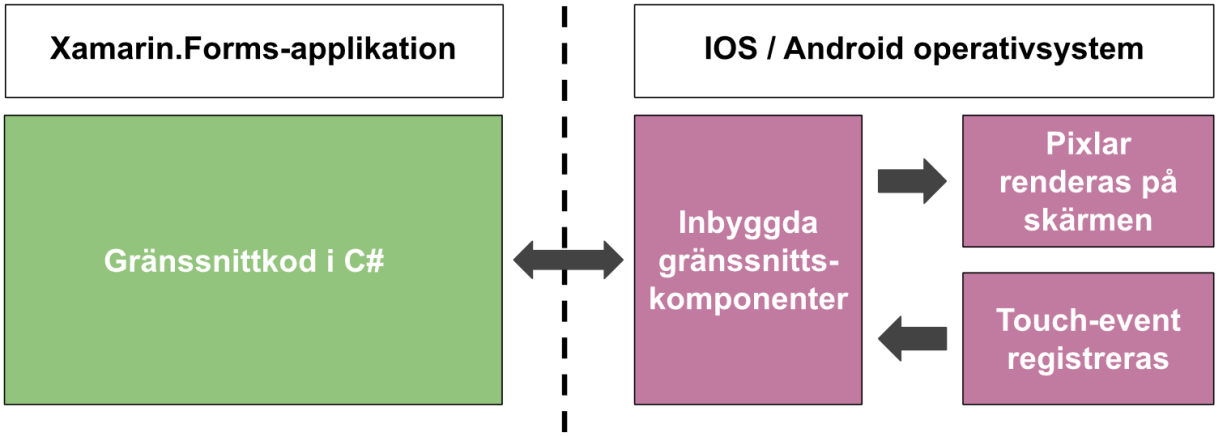
2.4.2 Xamarin.Forms

Xamarin.Forms släpptes 2014 och är en påbyggnad på tidigare nämnda ramverk från Xamarin [25]. Det som skiljer Xamarin.Forms från Xamarin.IOS/Xamarin.Android är att utvecklare ges möjlighet att dela en ännu större del av kodbasen då även koden för det grafiska gränssnittet är gemensam för både Android och IOS i Xamarin.Forms (se figur 2) [25]. Detta gör att utvecklarna inte behöver bygga *två* gränssnitt, ett för varje operativsystem, utan kan bygga *ett* enda som sedan används av båda operativsystem. Något som både underlättar och påskyndar utvecklingsprocessen.



Figur 2 - Arkitektur för Xamarin.Forms.

Precis som i Xamarin.Android/Xamarin.IOS så byggs gränssnitt i C#-kod som sedan kopplas till de inbyggda komponenter för gränssnitt som finns i Android och IOS, men i Xamarin.Forms så binder en och samma kod inte bara till komponenter i ett av operativsystemen, utan till motsvarande komponenter i båda, se figur 3.

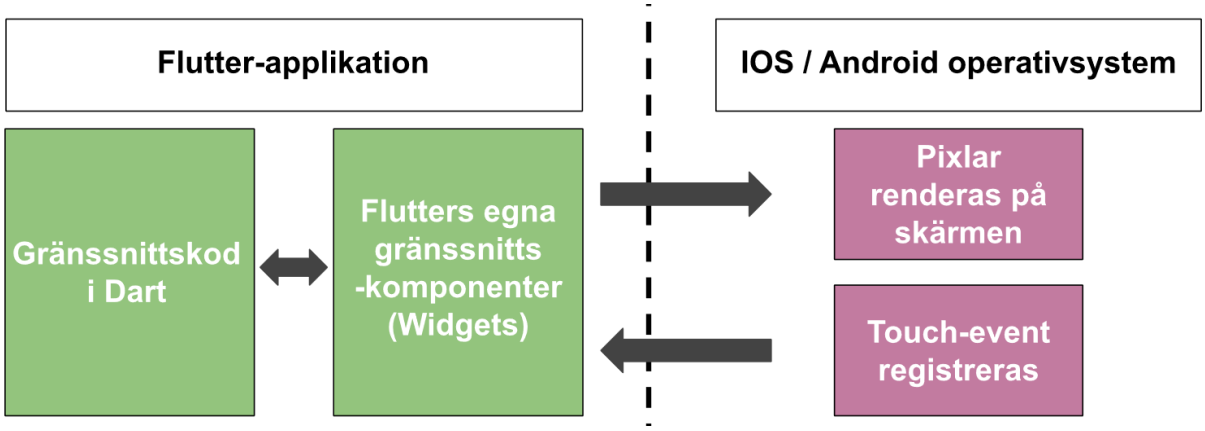


Figur 3 - Xamarin.Forms arkitektur för gränssnitt.

Den smidigare utvecklingsprocessen, med nästan helt delad kodbas mellan IOS och Android, är den allra största fördelen med Xamarin.Forms; detta sker dock på bekostnad av sämre prestanda vad gäller uppstartstid och vissa krävande operationer [26].

2.4.3 Flutter

Flutter, som släpptes i skarp version år 2018 [27], är Googles cross-platform-ramverk som använder sig av programmeringsspråket Dart för den gemensamma kodbasen [28]. Precis som med Xamarin.Forms så kan all skriven kod i Flutter, både för affärslogik och gränssnitt, delas mellan IOS och Android. Däremot åstadkommer Flutter den delade koden för gränssnitt på ett helt annat sätt. Istället för att skriva kod som binder till de inbyggda komponenterna i IOS och Android så väljer Flutter att rendera helt egna komponenter för gränssnitt, se figur 4 [29].

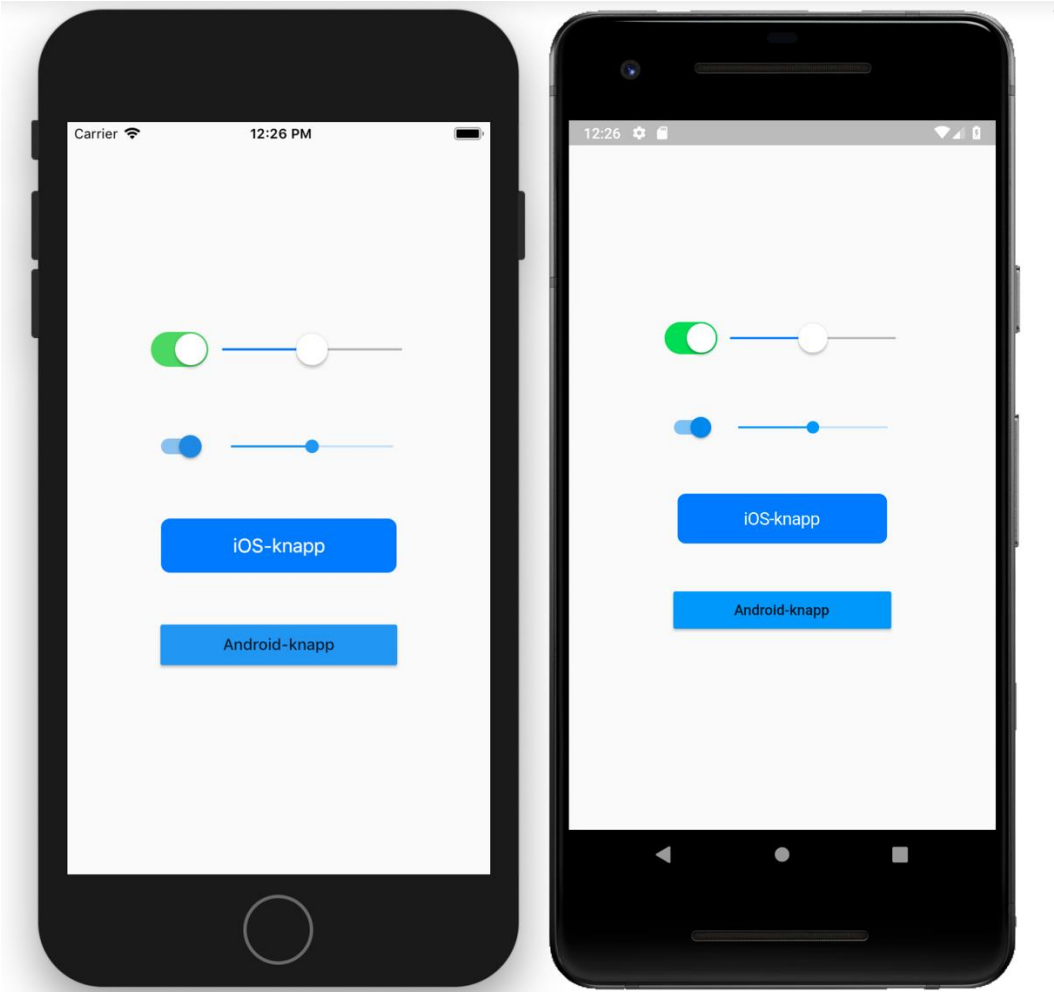


Figur 4 - Flutters arkitektur för gränssnitt.

Detta gör att alla komponenter som kan skapas i Flutter kan renderas på exakt samma sätt på alla enheter där Flutter kan köras, oberoende av operativsystem eller annat som kan tänkas styra stilen på gränssnittet för den enheten.

Dessa egna komponenter kallas för widgets och är en fundamental byggsten i Flutter [30]. Alla gränssnittskomponenter i Flutter är i någon form en typ av widget. En widget kan representera allt ifrån en knapp eller en layout till en animation, men de har alla gemensamt att de skall vara små och självständiga enheter vars utseende och beteende sedan kan skräddarsys i detalj. All design av gränssnitt bygger sedan på att placera widgets inne i widgets för att uppnå mer komplexa strukturer. I det bibliotek som följer med Flutter som standard finns många widgets som är exakta kopior av komponenter som finns tillgängliga i IOS och Android, såsom knappar och dialogrutor. I figur 5 kan vi se ett exempel på detta; en IOS- och en Androidenhet som båda

renderar två av/på-knappar, två dragreglage samt två knappar varav en av varje följer IOS respektive Androids stilguide för utseende och beteende.



Figur 5 *– IOS (vänster) och Android (höger) som renderar samma gränssnitt med Flutter.*

En annan nämnvärd funktion Flutter erbjuder är något som kallas *hot reload*, vilket innebär att man under utvecklingen kan uppdatera sin kod och se förändringen på en enhet på bara någon sekund, utan att förlora programmets tillstånd [31]. Detta fungerar genom att programmeringsspråket Darts VM (virtuell maskin) kan kompilera och ladda in endast den kod som ändrats medan programmet fortfarande exekveras. Detta möjliggör att utvecklare snabbt kan göra små justeringar i sina program utan att behöva vänta på att programmet ska kompileras från grunden.

2.4.4 React Native

React Native är ett cross-plattform-ramverk utvecklat av Facebook som släpptes år 2015 [32]. Ramverket är baserat på React, vilket är Facebooks JavaScript-bibliotek som används för att bygga användargränssnitt, framförallt för hemsidor. Koden skrivs i JavaScript, vilket är tätt förknippat med webbutveckling.

React Native har ett liknande tillvägagångssätt som Xamarin.Forms för att skapa gränssnitt i IOS och Android, ramverket använder sig av JavaScript för att binda till de inbyggda komponenter som finns i iOS och Android så att utvecklare kan rendera motsvarande komponenter på båda operativsystemen med en gemensam kodbas [33].

Precis som med Flutter (Dart) finns det möjlighet att göra hot reloads vilket innebär att applikationen kan fortsätta att köras och behålla sitt nuvarande tillstånd även när delar av den kompileras om [34].

När en React Native-applikation körs så tolkas dess JavaScript-kod med hjälp av en JavaScript-motor; JavaScriptCore [35]. På Android blir koden dessutom JIT-kompilerad, vilket medför bättre prestanda (jämfört med tolkning) om koden innehåller delar som upprepas ofta.

2.4.5 Ionic

Ionic är ett cross-plattform-ramverk som först släpptes 2014 och utvecklas av ett företag med samma namn (f.d. Drifty) [36]. Ramverket är en vidaretuveckling av Apache Cordova (Tidigare PhoneGap) [37]. Det bygger på ”web components” som är ett standardiserat system för att arbeta med HTML-element i olika webbläsare [38]. Det innebär att man kan återanvända samma HTML-element i alla webbläsare eller andra plattformar som använder JavaScript och/eller HTML och samtidigt lita på att de kommer att se ut och fungera likadant överallt [39].

Applikationer byggda med Ionic kodas i JavaScript, HTML samt CSS. De är baserade på webbteknologier och körs egentligen inuti en webbläsare på telefonen (WKWebView på IOS och Web View for Android på Android) [40]. Koden JIT-kompileras både på Android och IOS.

Ionic har vissa likheter med tidigare nämnda ramverk; likt Flutter använder Ionic inte de inbygga komponenterna i IOS och Android för att bygga gränssnitt och precis som i React-Native så är det primära utvecklingsspråket JavaScript.

2.4.6 Android (native)

Androidutveckling med Androids officiella SDK kan göras med Java, Kotlin samt C/C++ [41]. Applikationerna kan kompileras med antingen AOT eller JIT, alternativt en kombination av båda sedan Android 7.0 och framåt [42]. I Android Studio 2.3 eller senare finns en funktion som kallas *instant run* som möjliggör att man, under utvecklingsprocessen, kan uppdatera koden i en applikation utan att kompilera om hela applikationen, vilket liknar hot reload i Flutter och React-Native [43].

2.4.7 IOS (native)

Utveckling mot IOS med Apples officiella SDK sker i huvudsak med programmeringsspråken Swift och dess föregångare Objective-C, men man kan även skriva kod som inte berör applikationens UI med C/C++ [44]. Applikationerna är alltid AOT kompilerade [45].

3. Metod

Projektet har bestått av två primära delar.

Den första delen är en prestandajämförelse, med fokus på uppstartstid och installationsstorlek, mellan ramverken Xamarin.Forms, Xamarin och Flutter. Denna del är i sin tur uppdelad i två steg. Det första steget bestod av att ta fram en enkel applikation för varje ramverk som sedan användes för att etablera de optimala kompileringsparametrarna för respektive ramverk. Det andra steget bestod av att ta fram tre olika applikationer för respektive ramverk med de tidigare etablerade optimala kompileringsparametrarna. Dessa applikationer har sedan analyserats för att ge en grundläggande förståelse för hur varje ramverk presterar.

Den andra delen består av en djupdykning i ramverket Flutter genom utveckling av en prototypapplikation som imiterar en del av den funktionalitet som företagets nuvarande applikation erbjuder.

3.1 Grundläggande prestandamätningar

För att kunna jämföra prestandaskillnader mellan de olika ramverken har en metodik utarbetats utifrån följande tankar och idéer.

För att förstå hur olika kompileringsparametrar påverkar både uppstartstid och installationsstorlek för varje ramverk har först en enkel ”Hello World”-applikation tagits fram för varje ramverk. Dessa applikationer har kompilerats med alla tillgängliga kombinationer av parametrar för respektive ramverk. Detta för att avgöra vilka kombinationer av kompileringsparametrar som ger bäst resultat i form av uppstartstid och installationsstorlek för varje ramverk. Uppstartstid har här ansetts vara den viktigaste prestandafaktorn, följt av installationsstorlek. Som referens har också native-applikationer tagits med i mätningarna för att ge en bild av vad som bör ses som det optimala scenariot för respektive operativsystem.

Det slutgiltiga steget i prestandamätningarna mellan ramverken var att närmare undersöka uppstartstid och installationsstorlek för olika typer av applikationer. För att genomföra detta togs tre olika applikationer fram för vart och ett av ramverken Xamarin.Forms, Xamarin och Flutter:

- ”Hello World” – visar texten ”Hello World” på telefonens skärm.
- ”Bild ”– laddar in en bild från telefonens minne och visar den på skärmen.
- ”JSON lista” – laddar in en JSON-fil från telefonens minne med 1000 namn och renderar namnen som en rullbar lista.

Applikationerna är tänka att vara avskalade men samtidigt utföra uppgifter som är vanligt förekommande i mobila applikationer. Detta för att ge en grundläggande bild av de olika ramverkens prestandamässiga för- och nackdelar.

3.1.1 Mätmetodik

För att få så rättvisa mätresultat som möjligt och samtidigt säkerställa att testerna kan upprepas på ett kontrollerat sätt är det viktigt testerna utförs på ett korrekt sätt.

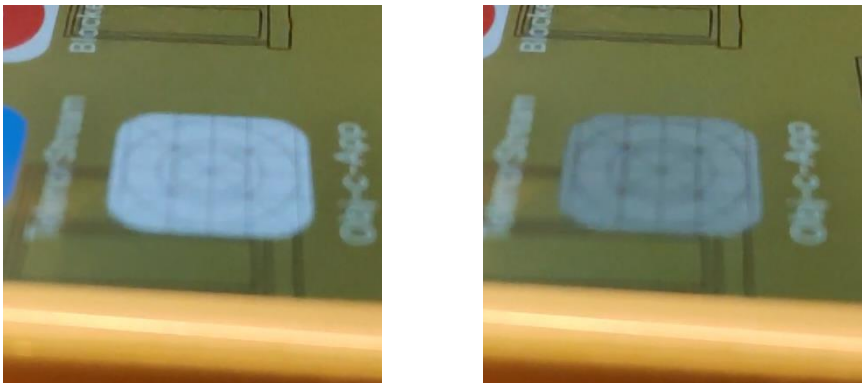
Både IOS och Android har verktyg i sina officiella IDEs (Xcode resp. Android Studio) för att mäta uppstartstid av applikationer, men dessa verktyg ger inte jämförbara resultat mellan olika ramverk och applikationstyper. Istället för att använda dessa verktyg så har video spelats in på förloppet när applikationerna har startats upp på testenheterna. Genom att analysera varje videoklipp, bildruta för bildruta, kan tiden mätas från att applikationen startas tills att den helt laddats in. Detta leder till att de resulterande uppmätta tiderna faktiskt motsvarar vad en verklig användare skulle uppleva.

Denna metod har använts i samtliga mätningar av uppstartstider med ett undantag; de initiala mätningarna där kompileringsparametrar jämfördes på Android. Dessa mätningar, som utfördes för att etablera vilka kompileringsparametrar som skulle användas för resterande tester, gjordes istället med verktyget *Logcat* i Android Studio. Där kan ett *displayed*-värde utläsas, som visar hur lång tid det gått mellan att applikationen startas till dess att den första bildrutan från applikationen visas [46]. Detta undantag gjordes för att spara tid, då mängden data som skulle mätas var så pass omfattande. Då mätningarna sker inom samma applikation och ramverk är resultaten rättvisande med denna metod inom detta specifika användningsområde.

Samtliga mätningar har utförts 10 gånger och därefter har ett medelvärde beräknats för att ta fram genomsnittlig uppstartstid för respektive applikation. För att göra testerna så jämlika mellan olika enheter som möjligt har det setts till att inga andra applikationer aktivt körs på testenheterna under testets gång. Mellan varje testförsök har applikationerna helt stoppats och

testenheternas *cache* av applikationerna tömts för att säkerställa att varje uppstart blir en så kallad *cold start* [47]. På Android innebär detta att göra en *force close* av applikationen mellan varje uppstart, medan det på IOS krävs en omstart av telefonen för att garantera en *cold start* av applikationen.

Videon som spelats in fångas med 240 bildrutor/s vilket innebär att varje bildruta motsvarar ca 4,17 ms. Telefonernas skärmar uppdateras endast i 60 Hz eller 60 bildrutor/s vilket innebär att bildrutorna uppdateras en gång varje 16,67 ms. Alltså kan uppstarstiden mätas med hög precision, från det att en användare startar applikationen genom att trycka på dess startikon på skärmen, tills dess att den första bildrutan från applikationens gränssnitt syns på skärmen. Det går att avgöra exakt när användaren trycker på startikonen på skärmen genom att ikonen blir mörkare när trycket har registrerats på IOS (se figur 6) och på Android har inställningen “visa skärmtryck” slagits på under “utvecklaralternativ” för att lättare kunna avgöra när ett tryck på skärmen har registrerats (se figur 7).



Figur 6- IOS startikon, före (t.v.) och direkt efter (t.h.) registrerat tryck på skärmen.

Figur 7 - Android startikon, den vita pricken visar ett registrerat tryck på skärmen.

3.1.2 Programvara och versioner

Vid utförandet av testerna användes följande mjukvara:

Xamarin.Forms 4.12.3.79
Xamarin.Android 9.1.5.0
Xamarin.IOS 12.2.1.12
Flutter v1.0.0
Microsoft Visual Studio 2017 Enterprise
Android Studio 3.3
ProGuard 6.0.3
JDK 1.8.0_152

3.1.3 Enheter vid testning

Testerna har utförts på följande hårdvara:

Oneplus 3 - Android 8.0.0
Samsung Galaxy Note 4 - Android 6.0.1
Iphone 6s - IOS 12.1.2

Allt videomaterial har spelats in med Xiaomi Pocophone F1.

3.1.4 Specifika inställningar för Visual Studio 2017 och Xamarin

Visual Studio och Xamarin erbjuder ett stort antal kompileringsparametrar som kan aktiveras för att uppnå olika syften i form av olika typer av prestandafördelar. Applikationer kan kompileras med AOT (Ahead of Time), ProGuard, LLVM (Low Level Virtual Machine) eller en kombination av dessa. Som standard är inga av dessa kompileringsalternativ valda och applikationen kompileras med JIT (Just in Time) utan några kompletterande parametrar. Inställningarna finns tillgängliga under “Properties” för Android-projektet, det skall dock tilläggas att inställningar såsom AOT och LLVM inte finns tillgängliga på alla Visual Studio-licenser. Alla applikationer är kompilerade under “Release”-läge.

3.1.5 Specifika inställningar för Android Studio och Flutter

Android Studio och Flutter erbjuder inte lika många kompileringsparametrar utan all kompilering som utförs är AOT oavsett målplattform. ProGuard erbjuds och kan aktiveras genom att använda inställningar i projektets “build.gradle”-fil enligt figur 8

```
buildTypes {
    release {
        signingConfig signingConfig.release

        minifyEnabled true
        useProGuard true

        proguardFiles getDeafultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
  }
}
```

Figur 8 - Kompileringsparametrar för Android Studio och Flutter.

minifyEnabled och useProguard sätts båda till “true” i de fall där applikationerna kompileras med Proguard påslaget. I de fall där Proguard inte inkluderats är båda dessa satta till “false”. I IOS-miljön finns det inga kompileringsinställningar att tillgå utan där är applikationerna kompilerade med standardinställningarna som innefattar AOT-kompilering. Proguard finns inte tillgängligt i IOS-miljön.

3.2 Prototypapplikation i Flutter

För att också undersöka utvecklingsprocessen i Flutter och ramverkets mognad i mer detalj så består en betydande del av projektet av utvecklingen av en prototypapplikation. Detta i syfte att utvärdera om ramverket Flutter kan erbjuda den funktionalitet som krävs för att kunna skapa en applikation liknande den som företaget använder idag. Genom utvecklingen av prototypapplikationen undersöktes också vilka andra eventuella fördelar ett byte av ramverk skulle kunna ge företaget.

4. Konstruktion av prototypapplikationen

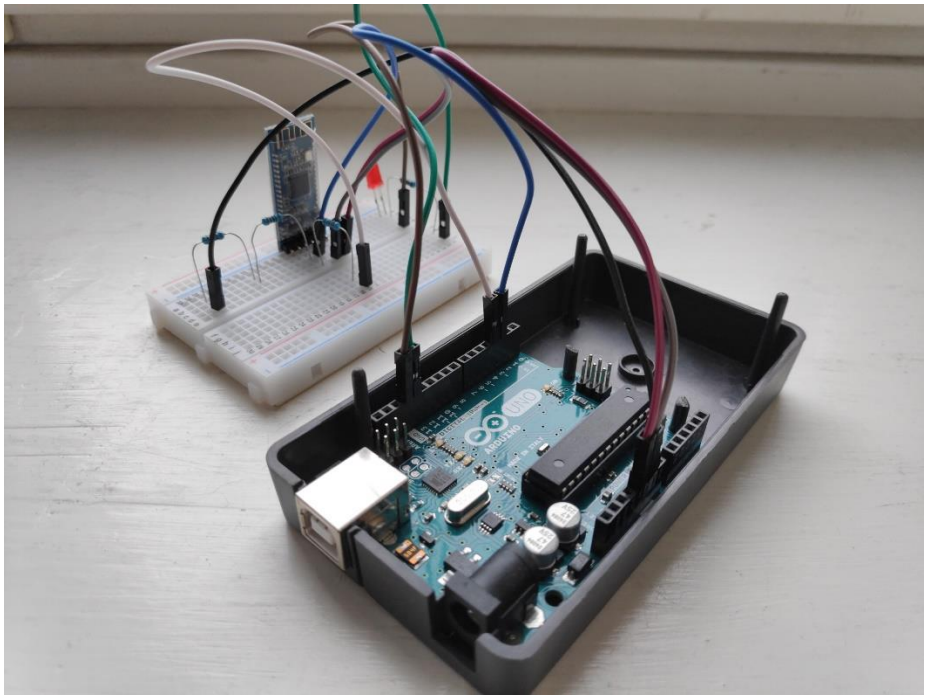
I början av projektet skapades en kravspecifikation som utgjort grunden för den funktionalitet som inkluderats i applikationen.

Kravspecikation för protypapplikationen:

- Möjlighet att kommunicera med företagets Bluetooth-enheter utan att modifiera dessa.
- Automatisk uppkoppling mot närliggande Bluetooth-enhet vid uppstart av applikationen.
- Styra Bluetooth-enheter på ett liknande sätt som i den befintliga applikationen.
- Möjlighet att kommunicera med företagets backend för att hämta data om befintliga konton och platser.
- Möjlighet att byta mellan olika platser för att kunna styra dessa.

Tanken bakom kravspecifikationen för prototypen var att inkludera all den grundläggande funktionalitet som krävs för att styra befintliga system som redan är konfigurerade.

Till en början konstruerades en enkel applikation med möjligheten att söka efter närliggande BLE-enheter och ansluta till dessa. För att åstadkomma detta användes ett tredjepartsbibliotek vid namn FlutterBlue, som är ett bibliotek skrivet i Dart, designat att förenkla Bluetooth-kommunikation i Flutter-applikationer [48]. I detta utförande hade applikationen möjlighet att manuellt skicka kommandon till den anslutna BLE-enheten. De initiala testerna att skicka och motta data skedde i huvudsak mot en Arduino-enhet (se figur 9) som konfigurerats för att motta meddelanden och samtidigt skriva ut det mottagna meddelandet i Arduinos seriella konsol. Som ett komplement till utskriften av meddelanden anslöts också en LED som programmerades att tändas och släckas vid mottagen 1 respektive 0. Arduino-enheten konfigurerades också så att den löpande skickade ut nya meddelanden för att på sätt testa om det från prototypapplikationen var möjligt att prenumerera på förändringar och nya meddelanden från Bluetooth-enheten.



Figur 9 - BLE-modul och röd LED (t.v) med tillhörande Arduino Uno (t.h.).

Efter att ha säkerställt att den grundläggande Bluetooth-funktionaliteten fungerade tillfredsställande, fortsatte arbetet genom att implementera funktionalitet för anslutning och sändning av data till en av företagets BLE-enheter. Företagets BLE-enheter använder sig av kryptering för att autentisera applikationen mot BLE-enheten. För att möjliggöra autentiseringen implementerades den nödvändiga krypteringslogiken i prototypapplikationen med hjälp av ett tredjepartsbibliotek. Efter att ha verifierat att autentiseringen av applikationen fungerat som tänkt behövde även den data som skickas till BLE-enheten krypteras. Denna logik implementerades för att göra det möjligt att skicka kommandon till BLE-enheten för att t.ex. ställa in lampornas ljusstyrka.

I kraven för applikationen ingår också att kommunicera med företagets befintliga backend. Målet med detta var att ha möjlighet att autentisera användarkonton mot backend och även hämta den information som möjliggör kommunikation med företagets hårdvara. Tack vare ett tredjepartsbibliotek kunde inloggningen och autentiseringen mot företagets backend implementeras relativt enkelt.

Den data som hämtas från backend levereras i JSON-format, vilket är ett vanligt förekommande format i applikationer som skickar och tar emot data över webben. All den data som hämtas från

databasen är dock inte relevant för att endast styra enheter, därför gjordes valet att skapa en egen JSON-struktur i applikationen för att på ett enkelt sätt kunna hantera och spara undan just den data som är nödvändig för applikationens användningsområde.

Efter en första inloggning i applikationen, där data om det inloggade kontot och dess tillhörande platser hämtas från backend, skrivs den relevanta datan till en fil som sparas på telefonen. Detta för att applikationen ska kunna starta snabbare vid nästkommande uppstartstillfällen, då all information om användaren läses in lokalt istället för att hämtas från backend.

För att underlätta för användaren, ansluter prototypapplikationen efter uppstart automatiskt till den första aktiva Bluetooth-enheten som tillhör någon av användarens förkonfigurerade platser. Har användaren inte aktiverat Bluetooth-funktionaliteten på sin telefon varnas användaren om detta när applikationen försöker söka efter enheter i närheten. Vid aktivering av Bluetooth startas sökningen efter BLE-enheter och anslutningsprocessen fortsätter som vanligt.

När det gäller styrning av Bluetooth-enheter är en viktig egenskap för applikationen att det finns möjlighet att skicka kommandon till andra enheter än den specifika Bluetooth-enhet den är uppkopplad mot just för tillfället. Under prototypapplikationens utveckling implementerades stöd för att hantera ett obegränsat antal rum innehållande ett obegränsat antal enheter, detta för att inte begränsa användaren och ge stöd åt alla de konfigurationer som är möjliga i systemet. Även funktionalitet för att möjliggöra styrning av lampor som är anslutna till andra enheter än den aktuella uppkopplade Bluetooth-enheten, via mesh-nätverket, implementerades i detta stadium.

Som ytterligare ett komplement implementerades också funktionalitet för att kunna lyssna på förändringar i mesh-nätverket. Tanken bakom detta är att om en telefon styr lampor på den uppkopplade platsen så rapporteras detta till alla uppkopplade telefoner. Telefonerna som *inte* utfört förändringen i ljusstyrka justerar då automatiskt sina reglage för att återspegla de värden som lamporna i systemet har för tillfället. Detta gjordes framförallt för att efterlikna den ursprungliga applikationens beteende och för att göra styrningen mellan applikationerna kompatibel.

I detta stadium var prototypapplikationen frikopplad från företagets applikation till den grad att användaren har möjlighet att logga in på sitt befintliga konto, ansluta mot sina BLE-enheter och kontrollera belysningen på ett liknande sätt som är möjligt i företagets egen applikation. Prototypen följer företagets implementation av hur styrningen av lamporna fungerar, detta för att den befintliga applikationen och prototypapplikationen ska kunna styra samma plats utan att några missförstånd uppstår i hur de olika applikationerna tolkar den data som utläses från mesh-nätverket.

5. Resultat av tester

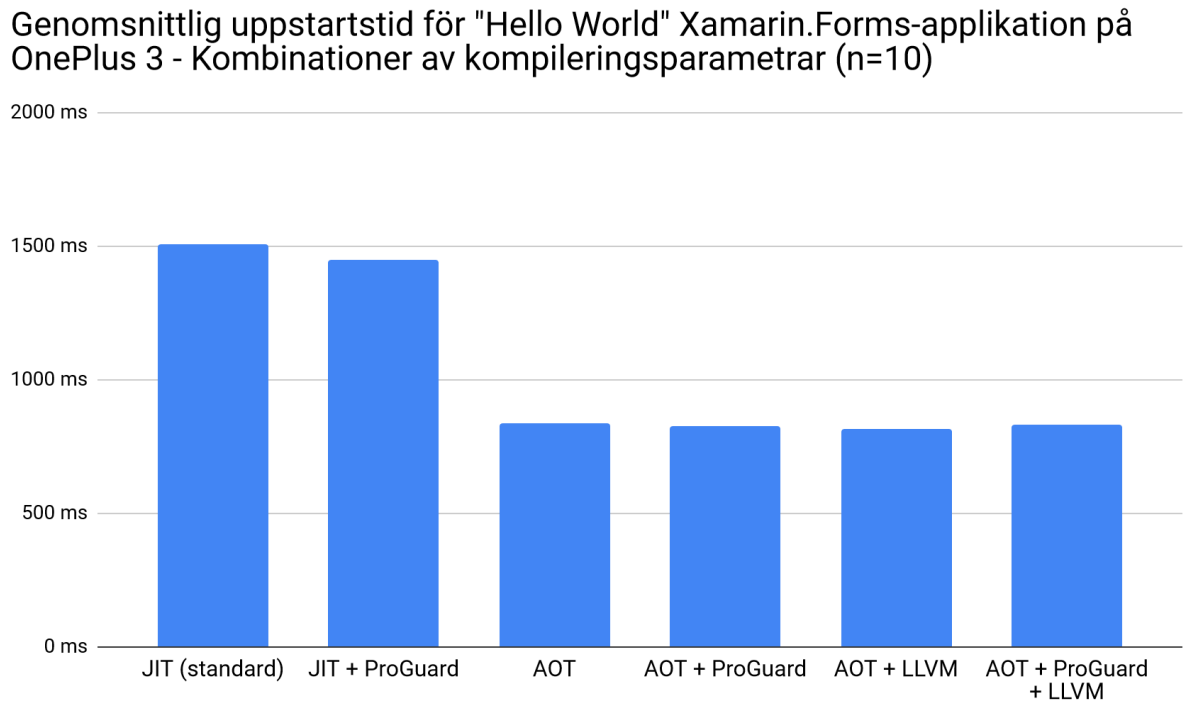
I detta kapitel behandlas resultaten av de tester som utförts mellan ramverken. Kapitlet är uppdelat i två huvuddelar där resultaten för testerna på Android och IOS presenteras separat.

5.1 Android

På Android-plattformen observerades stora skillnader inbördes inom ramverken där kompileringssparametrarna kan vara en viktig faktor som påverkar hur en applikation presterar. Även mellan ramverken uppvisades stora skillnader med hänsyn till uppstartstid och installationsstorlek för de applikationer som kompilerats med de optimala kompileringsparametrarna. Observera att endast mätningar från OnePlus 3-telefonen presenteras i detta kapitel, detta först och främst för att göra resultaten mer lättläsliga men också p.g.a. att samma proportioner mellan ramverkens resultat även återfinns vid tester utförda på Samsung Galaxy Note 4.

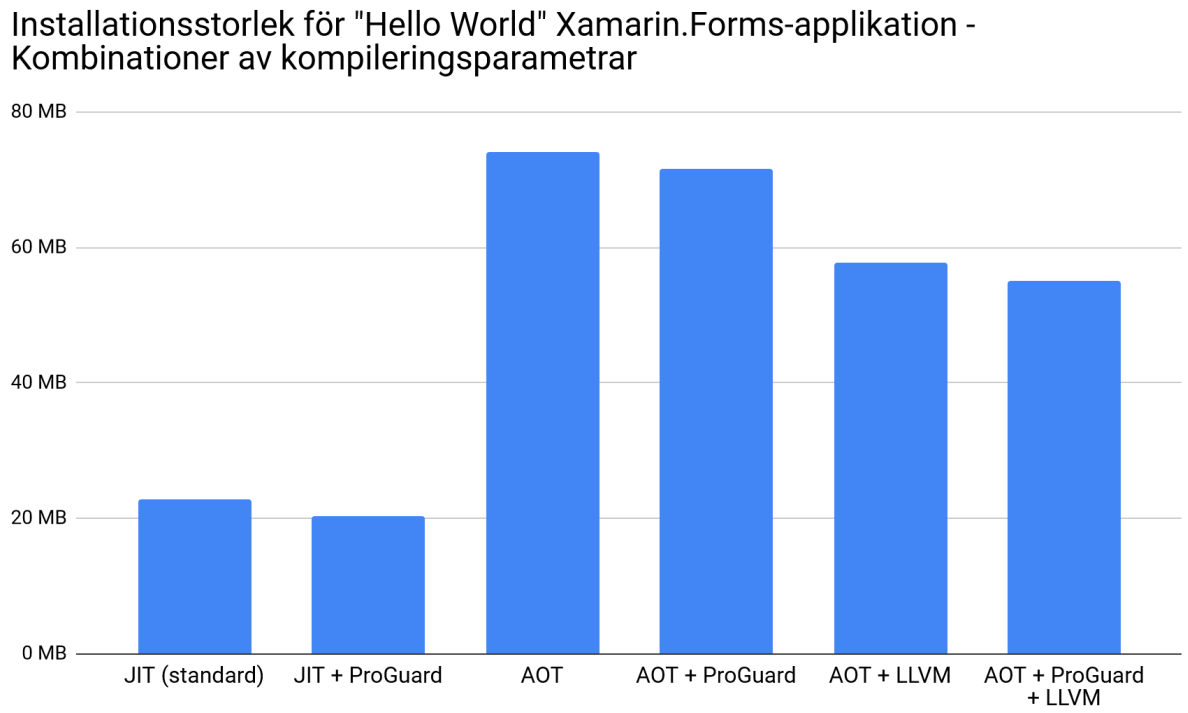
5.1.1 Etablering av kompileringsparametrar

Vid genomförandet av det initiala testerna, vars syfte var att belysa de olika kompileringsparametrarnas för-/nackdelar för respektive ramverk, kunde tydliga mönster ses där resultaten från Android-telefonerna pekade på att AOT-kompilering ger stora fördelar när det kommer till uppstartstid för Xamarin och Xamarin.Forms (se figur 10). Dock verkar inte någon kombination av de övriga parametrarna ProGuard eller LLVM ha någon direkt inverkan på uppstartstiden så länge som AOT-kompilering är aktivt. De JIT-kompilerade applikationerna har en betydligt högre uppstartstid och inte heller här verkar ProGuard, vilken är den enda tillgängliga kompilationsparametern vid JIT-kompilering, ha någon större inverkan på uppstartstiden.



Figur 10 - Resultat från mätning av uppstartstid av ”Hello World”-applikationer i Xamarin.Forms med olika kompileringsparametrar.

När det gäller installationsstorleken för Xamarin.Forms-applikationerna ges dock en annan bild av kompileringsparametrarnas inverkan på resultaten. Installationsstorleken hos de JIT-kompilerade applikationerna är betydligt mindre än motsvarande AOT-kompilerade applikationer (se figur 11). Detta kan härledas till att telefonen sparar den färdiga maskinkoden för framtida exekvering när applikationen AOT-kompileras, vilket tar stor plats. I de fall där JIT-kompilering använts kompileras maskinkoden precis när den behövs för att sedan kasseras när applikationen stängs, vilket får en positiv effekt på applikationens installationsstorlek.



Figur 11 - Resultat från mätning av installationsstorlek med olika kompileringsparametrar för ”Hello World”-applikationer i Xamarin.Forms.

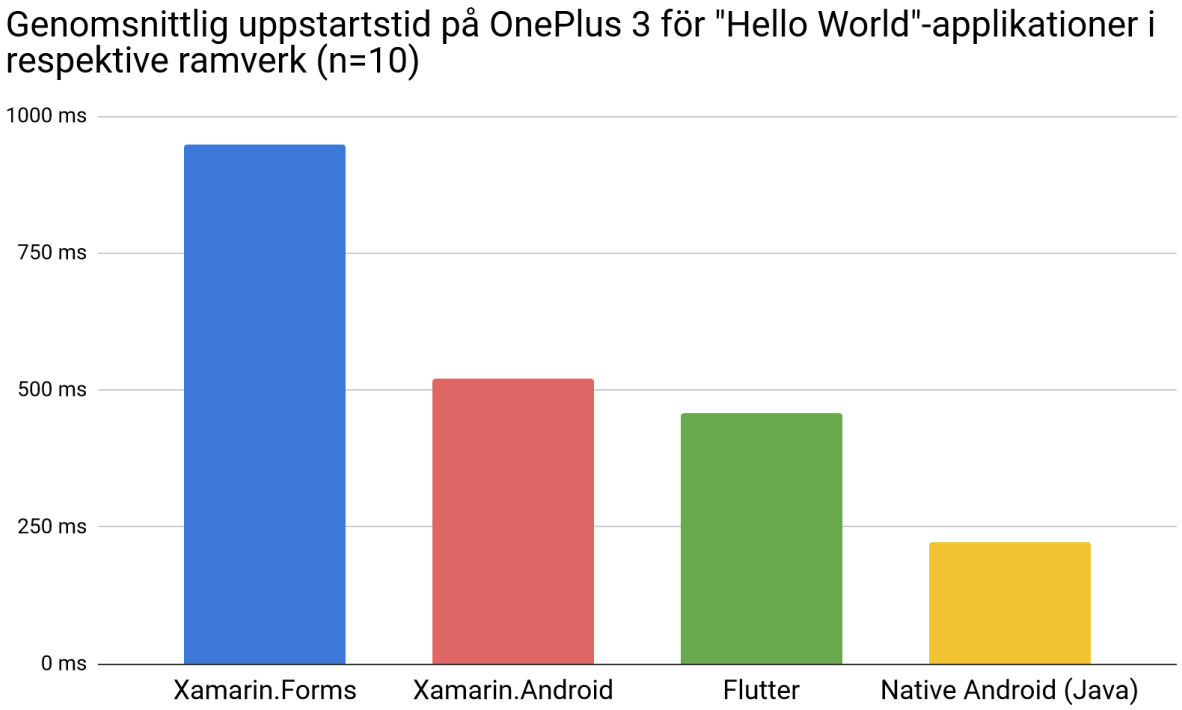
När endast installationsstorleken tas i beaktning upplevs också en skillnad mellan de kombinationer av kompileringsverktyg som finns tillgängliga.

Proguard har en marginell positiv effekt på installationsstorleken hos applikationerna oavsett om de kompilerats med JIT- eller AOT-kompilering. LLVM är den parameter som påverkar installationsstorleken mest påtagligt när den används. LLVM finns endast tillgängligt för AOT-kompilerade applikationer på Xamarin-plattformen.

Efter de initiala testresultaten analyserats fattades beslutet att använda både AOT, Proguard samt LLVM för resterande applikationer på Xamarin-plattformen. Det var den kombination som gav bäst resultat med hänsyn till både uppstartstid och installationsstorlek. För Flutter-applikationen stod valet mellan att använda sig av AOT-kompilering med eller utan Proguard. Då Proguard inte verkade ha någon negativ inverkan på applikationens prestanda i form av uppstartstid och installationsstorlek gjordes valet att inkludera Proguard i resterande Flutter-applikationer.

Som referens skapades också en native ”Hello World” Android-applikation. Detta först och främst i syfte att ge en insikt i hur det optimala scenariot ser ut för Android-plattformen när det kommer till de två prestandaparametrarna. Android-applikationen har precis som Flutter ett väldigt begränsat antal kompileringsparametrar bestående JIT-/AOT-kompilering med valet att slå på eller av ProGuard. Kombinationen AOT med ProGuard påslaget ansågs ge bäst resultat då uppstartstiden förbättrades samt att applikationens installationsstorlek krympte.

I figur 12 redovisas uppstartstiden för varje ramverk och dess respektive ”Hello World”-applikation. Applikationerna är kompilerade med de optimala kompileringsparametrar som ovan etablerats för respektive ramverk.

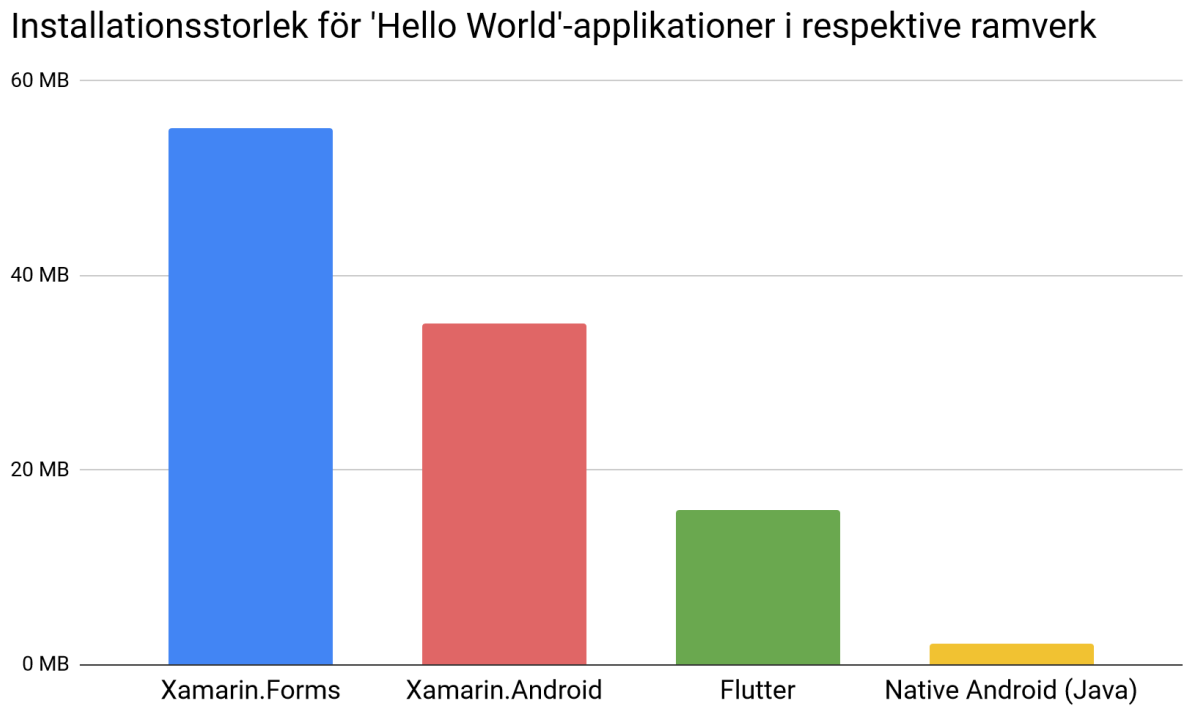


Figur 12 - Resultat från mätning av genomsnittlig uppstartstid för ”Hello World”-applikationer i respektive ramverk.

Som figur 12 visar har native Android-applikationen överlägset kortast uppstartstid. Flutter och Xamarin.Android har relativt jämförbara uppstartstider medan Xamarin.Forms har absolut högst uppstartstid.

Rörande installationsstorlek för ”Hello World”-applikationerna observeras även här stora skillnader mellan ramverken. I figur 13 redovisas installationsstorleken för ”Hello World”-

applikationerna ifrån respektive ramverk. Native Android-applikationen är åter igen den bäst presterande och har överlägset minst installationsstorlek. Xamarin.Forms har den absolut största installationsstorleken. Xamarin.Android och Flutter har betydligt mindre storlekar än Xamarin.Forms-applikationen men är fortfarande långt ifrån native-implementationen.



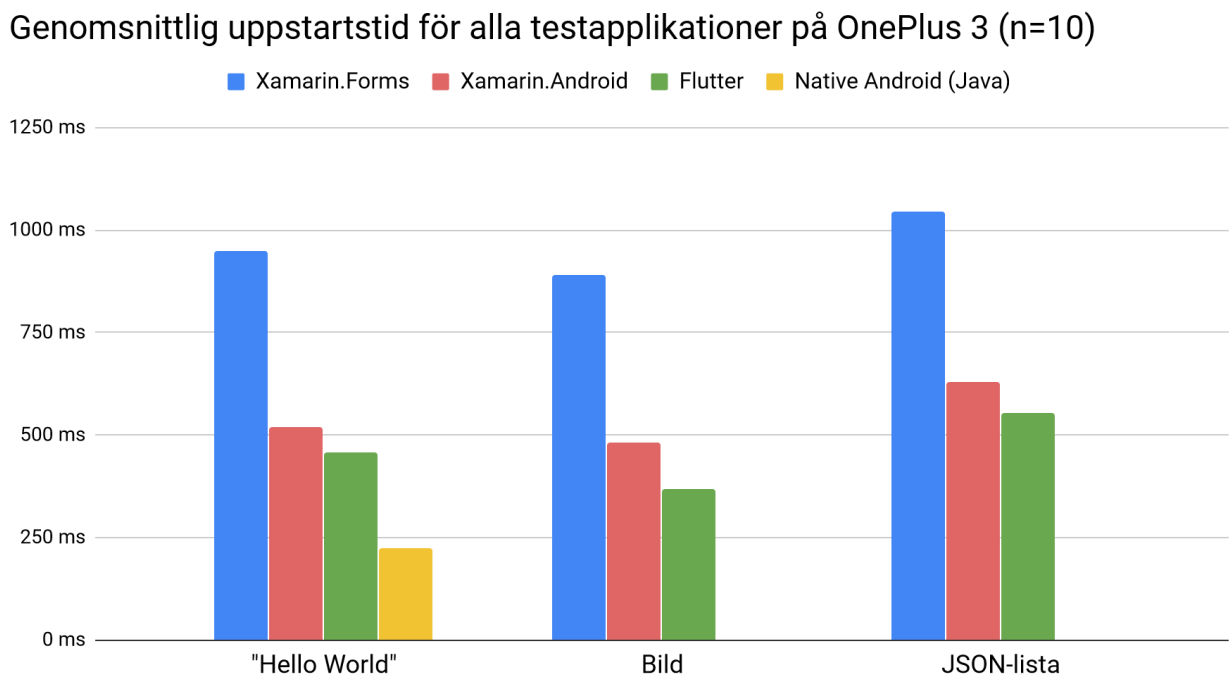
Figur 13 – Installationsstorlek för ”Hello World”-applikationer i respektive ramverk.

5.1.2 Övriga applikationer

Förutom de prestandamätningar som utförts på ”Hello World”-applikationerna utfördes även prestandamätningar på de andra två typerna av applikationerna som tidigare nämnts; ”Bild” och ”JSON lista”. Alla applikationer är här kompilerade med sina respektive optimala kompileringsinställningar som etablerats i de föregående mätningarna.

Bild-applikationen uppvisade liknande prestandaskillnader mellan ramverken som vid tidigare tester. Xamarin.Forms-varianten av bild-applikationen hade betydligt högre uppstartstid (se figur 14) jämfört med motsvarande applikationer skapade i Xamarin.Android och Flutter. Uppstartstiden för Xamarin.Forms-applikationen var här ca 1,8 ggr längre än motsvarande Xamarin.Android-applikation. Jämfört med Flutter-applikationen är uppstartstiden mer än

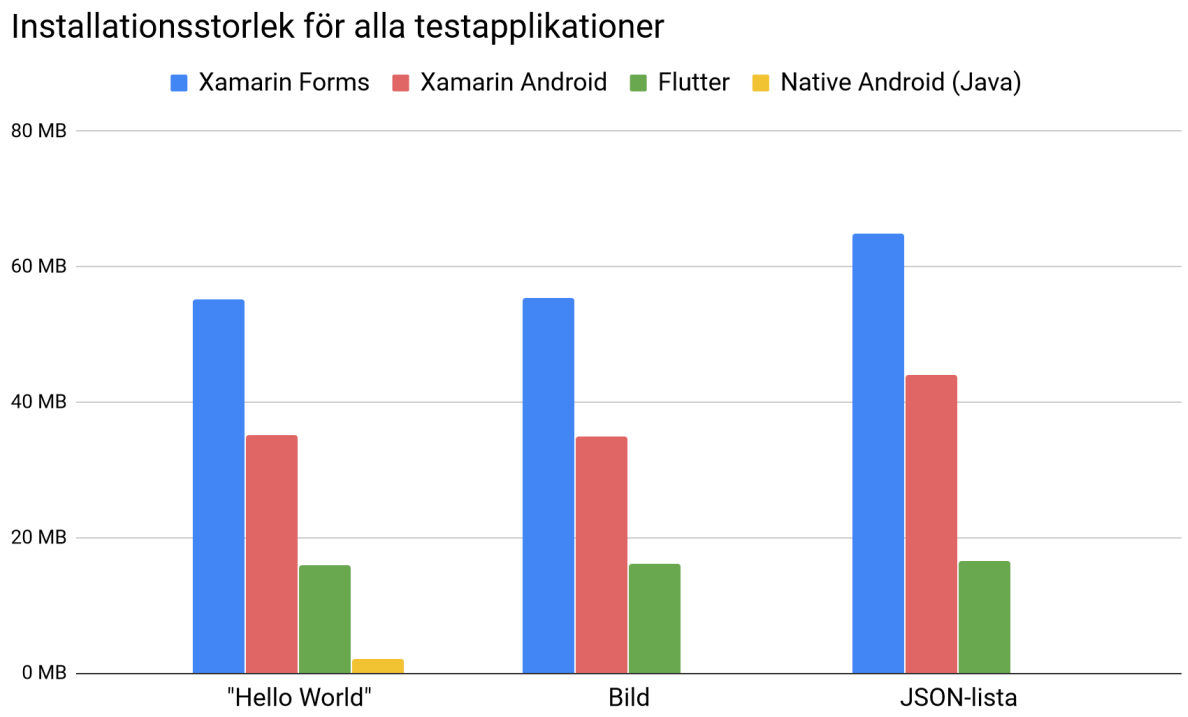
dubbelt så lång för Xamarin.Forms-applikationen. Något förvånande var Bild-applikationerna snabbare än ”Hello World”-applikationerna. Detta beteende observerades för alla ramverk.



Figur 14 - Översikt över uppstartstiden för alla testapplikationer.

Som typexempel på variation av mätdata för Android-applikationerna har Bild-applikationen skapad i Xamarin.Forms en maximal uppmätt uppstartstid på 954 ms och en lägsta uppmätt uppstartstid på 826 ms, detta jämfört med dess medelvärde på 888 ms. Variationskoefficienten för applikationens kompletta testserie är ca 4,5%, vilket är representativt för de mätningar som utförts på Android-plattformen. Inget av de testade ramverken eller applikationstyperna avviker speciellt när det gäller spridning av mätvärden.

Precis som i de tidigare mätningarna var det inte heller bara uppstartstiden där ramverken uppvisade stora skillnader i prestanda. Installationsstorleken är helt klart fortfarande till fördel för Xamarin.Android och Flutter jämfört med Xamarin.Forms (se figur 15).



Figur 15 - Installationsstorlek för alla testapplikationer.

Förutom ”Hello World”-applikationen och Bild-applikationen utfördes också mätningar på den typ av applikation som läser in 1000st JSON-objekt från en fil och visar dessa i en rullbar lista.

Även här uppvisas ungefär samma mönster som tidigare observerats mellan de olika ramverken. Uppstartstiden är återigen ca 1,8 ggr längre för Xamarin.Forms jämfört med dess Xamarin.Android-ekvivalent (se figur 14). Avståndet mellan Xamarin.Forms-applikationens och Flutter-applikationens uppstartstid krymper här en aning jämfört med tidigare applikationer och är här precis dubbelt så lång. Installationsstorleken för Xamarin.Forms och Xamarin.Android växer en aning, där den mest anmärkningsvärda skillnaden är att Flutter-applikationen är ¼ så stor som Xamarin.Forms-applikationen (se figur 15).

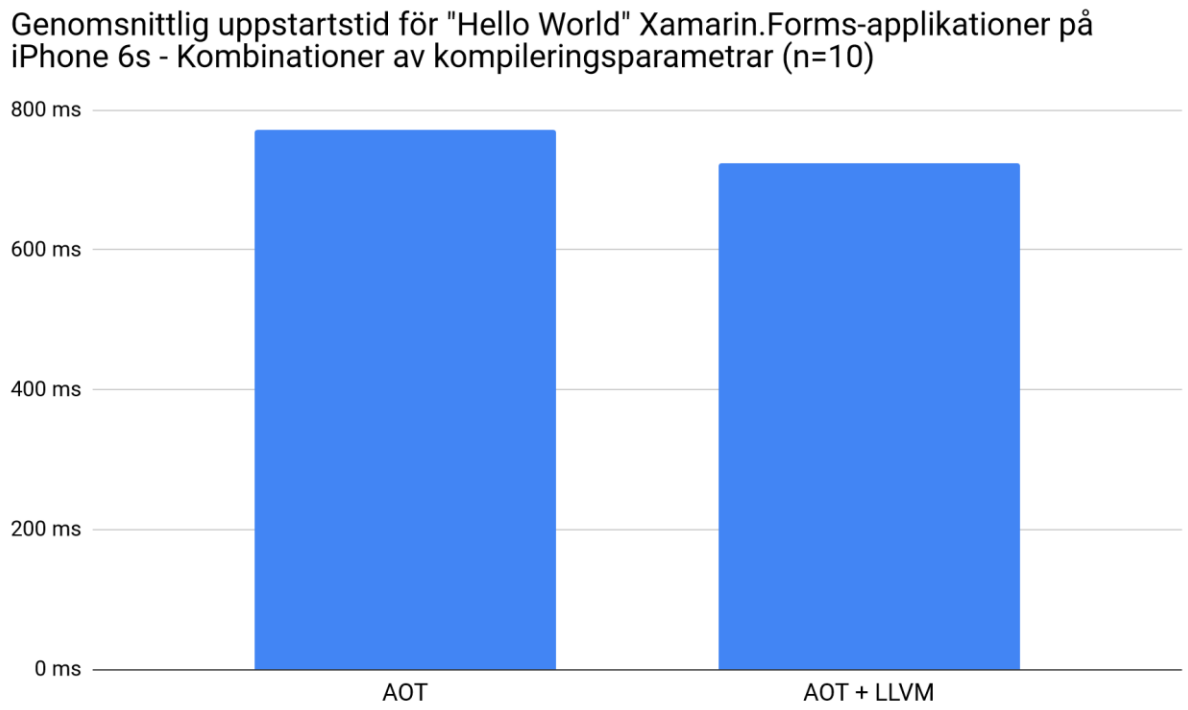
Alla testresultat pekar på att det finns stora prestandaskillnader mellan Xamarin.Forms och de andra två ramverken, Xamarin.Android och Flutter. Både Xamarin.Android och Flutter har genomgående lägre uppstartstider och mindre installationsstorlekar. Xamarin.Android och Flutter är betydligt mer jämlika med hänseende till de två prestandaaspekterna jämfört med Xamarin.Forms. Dock får alla ramverken se sig slagna av native-applikationen som är överlägset minst och snabbast.

5.2 IOS

IOS-plattformen uppvisade en betydligt mindre spridning av prestandan mellan ramverken, dessutom finns här inte lika många kompileringsparametrar att ta hänsyn till. Även mellan applikationstyperna uppvisades inga större skillnader i prestanda.

5.2.1 Etablering av kompileringsparametrar

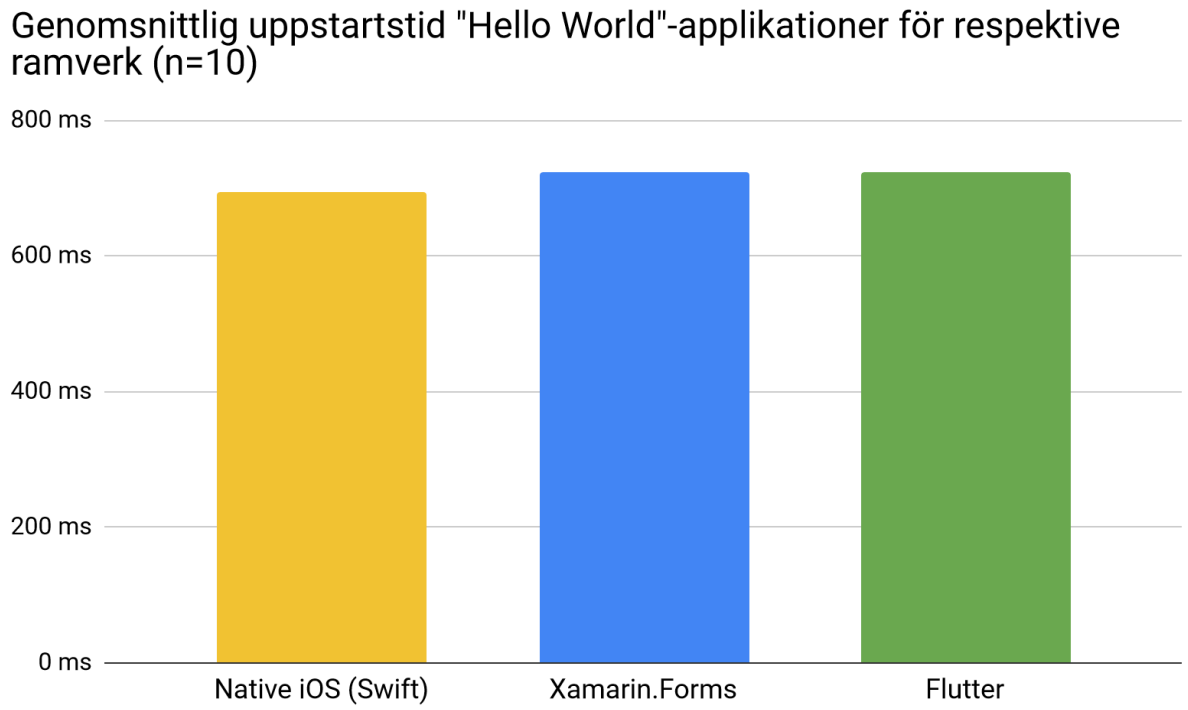
IOS har ingen inbyggd ProGuard eller motsvarande funktionalitet. Inte heller finns möjligheten att välja om koden ska kompileras med JIT-kompilering eller AOT-kompilering. AOT-kompilering är det enda alternativet på IOS-plattformen. Xamarin har dock har möjligheten att aktivera LLVM. De inledande testerna av ”Hello World”-applikationen skapad i Xamarin.Forms visade på att LLVM gav en marginell förbättring i uppstartstid (se figur 16).



Figur 16 - Genomsnittlig uppstartstid för ”Hello World”-applikation i Xamarin.Forms med LLVM på/av.

Ingen nämnvärd skillnad i installationsstorlek observerades med LLVM påslaget, men p.g.a. den förbättrade uppstartstiden gjordes valet att inkludera LLVM i resterande Xamarin.Forms-applikationer. I Flutter finns inga kompileringsparametrar att tillgå, endast AOT-kompilering är möjlig, utan ytterligare alternativ. Även för IOS-plattformen skapades en native-applikation för

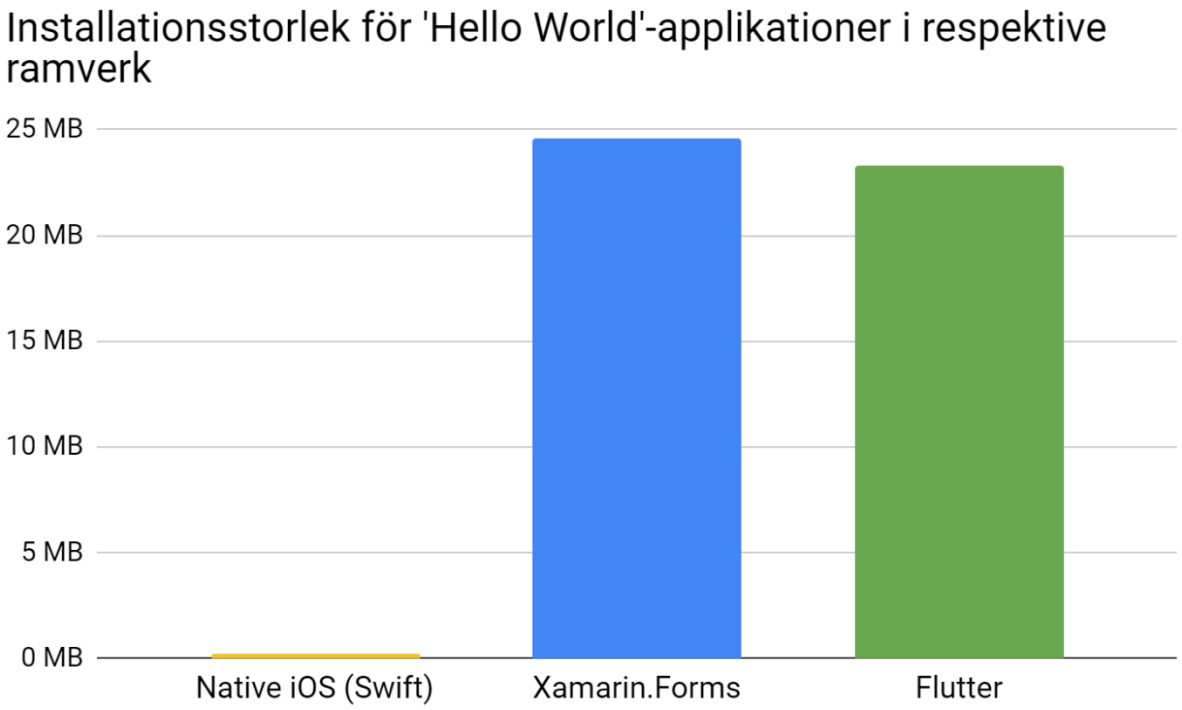
att visa på det optimala scenariot för plattformen med hänsyn till både uppstartstid och installationsstorlek. Precis som för Flutter saknas här också kompileringsparametrar och all kompilering är AOT. I figur 17 visas uppstartstiden för respektive ramverks ”Hello World”-applikation, med deras optimala kompileringsparametrar.



Figur 17 - Genomsnittlig uppstartstid för samtliga ”Hello World” IOS-applikationer.

”Hello World”-applikationerna uppvisade betydligt mindre skillnader vid mätning av uppstartstid jämfört med motsvarande Android-applikationer. Uppstartstiden för både Flutter-applikationen och Xamarin.Forms-applikationen uppmättes till exakt samma tid (se figur 17). Native-applikationen uppvisade en lite bättre uppstartstid än både Flutter- och Xamarin.Forms-applikationerna, men endast en marginell skillnad på ett fåtal millisekunder.

Även installationsstorleken var betydligt jämnare mellan ramverken, dock med undantaget att native IOS-applikationen är ca 1/100 så stor som både Flutter och Xamarin.Forms-applikationerna (se figur 18).



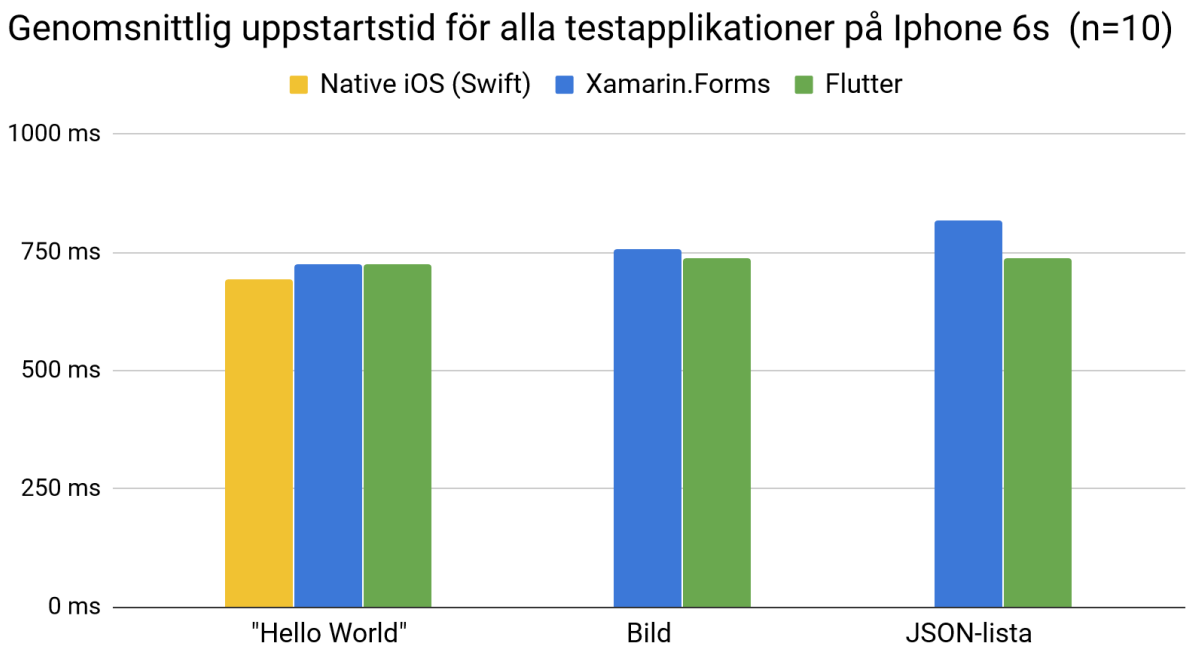
Figur 18 - Installationsstorlek för ”Hello World”-applikationer på IOS-plattformen.

5.2.2 Övriga applikationer

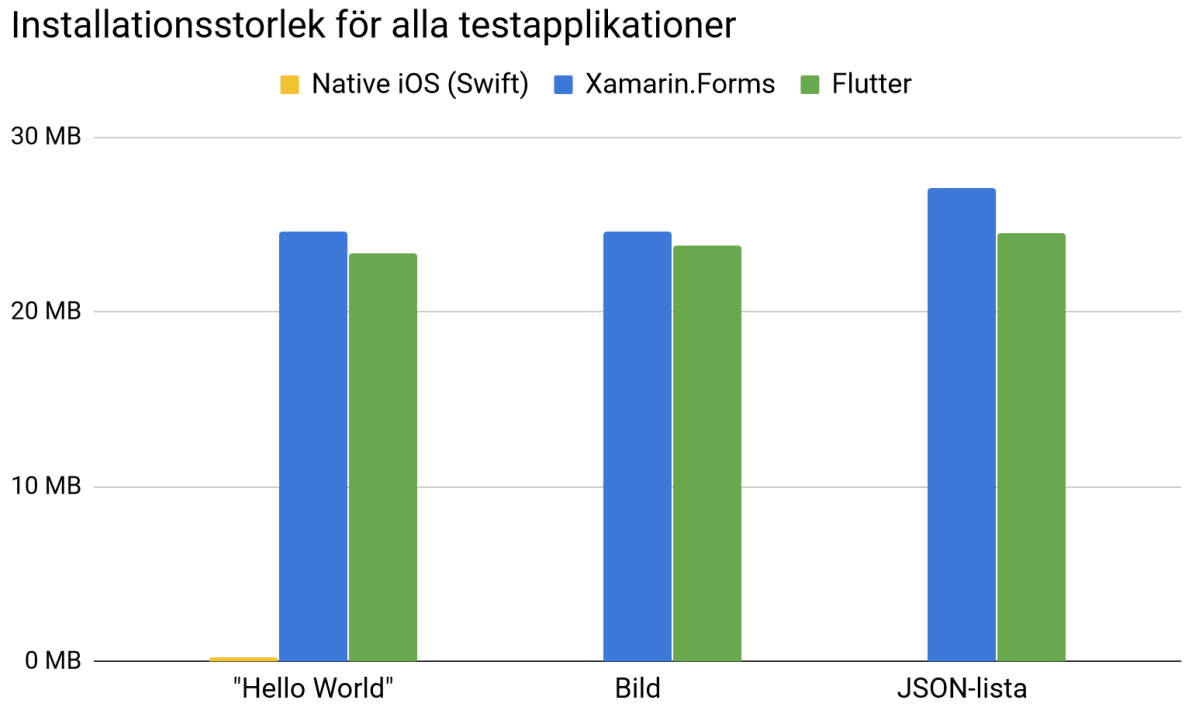
På samma sätt som för Android-plattformen utfördes också tester på de två andra applikationstyperna, ”Bild” och ”JSON lista”, för respektive ramverk.

Bild-applikationen är här marginellt snabbare i Flutter jämfört med Xamarin.Forms (se figur 19), men det är fortfarande en ytterst liten skillnad mellan ramverken på endast ett fåtal millisekunder. Även installationsstorleken (figur 20) skiljer sig endast marginellt mellan Flutter och Xamarin.Forms, även här till Flutters fördel.

JSON lista-applikationen uppvisar ungefär samma resultat mellan ramverken. Återigen har Flutter-applikationen en något lägre uppstartstid än sin Xamarin.Forms-ekvivalent (se figur 19). Även installationsstorleken (figur 20) uppvisar en liknande skillnad mellan Flutter och Xamarin.Forms, även denna gång till fördel för Flutter.



Figur 19 - Genomsnittlig uppstartstid för alla applikationer på IOS-plattformen.



Figur 20 - Översikt över installationsstorlek för alla ramverk och applikationer som testats på IOS-plattformen.

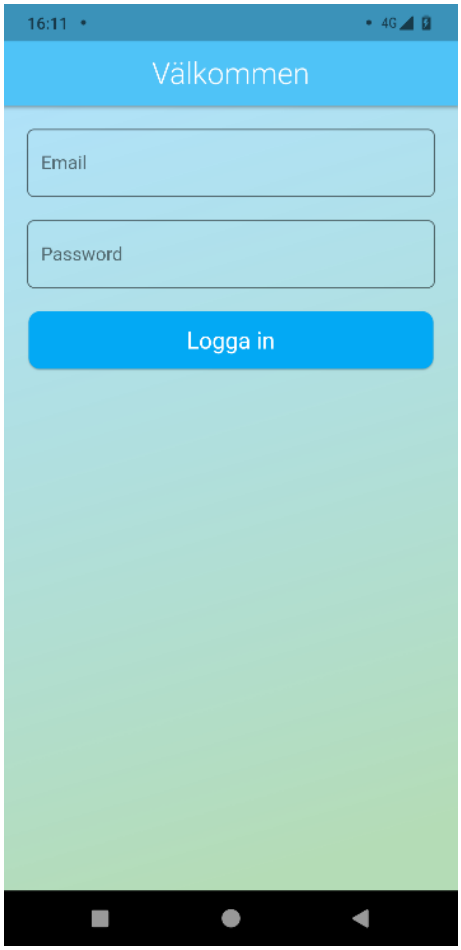
Som typexempel på variation av mätdata för en typisk IOS-applikation har Bild-applikationen skapad i Xamarin.Forms en högsta uppmätt uppstartstid på 759 ms och en minsta uppmätt uppstartstid på 730 ms, detta jämfört med applikationens genomsnittliga uppstartstid på 746 ms. Variationskoefficienten för applikationens kompletta testserie är ca 2%, vilket är representativt för de mätningar som utförts på IOS-plattformen. Inget av de testade ramverken eller applikationstyperna avviker speciellt när det gäller spridning av mätvärden.

Generellt sett presterar alla applikationer betydligt mer jämlikt på IOS än på Android oavsett ramverk. Sett till uppstartstid hade native-applikationen en väldigt liten förbättring jämfört med både Flutter och Xamarin.Forms. Den största skillnaden är installationsstorleken av IOS-applikationen jämfört med respektive ramverks applikationer.

6. Resultat av applikationens utveckling

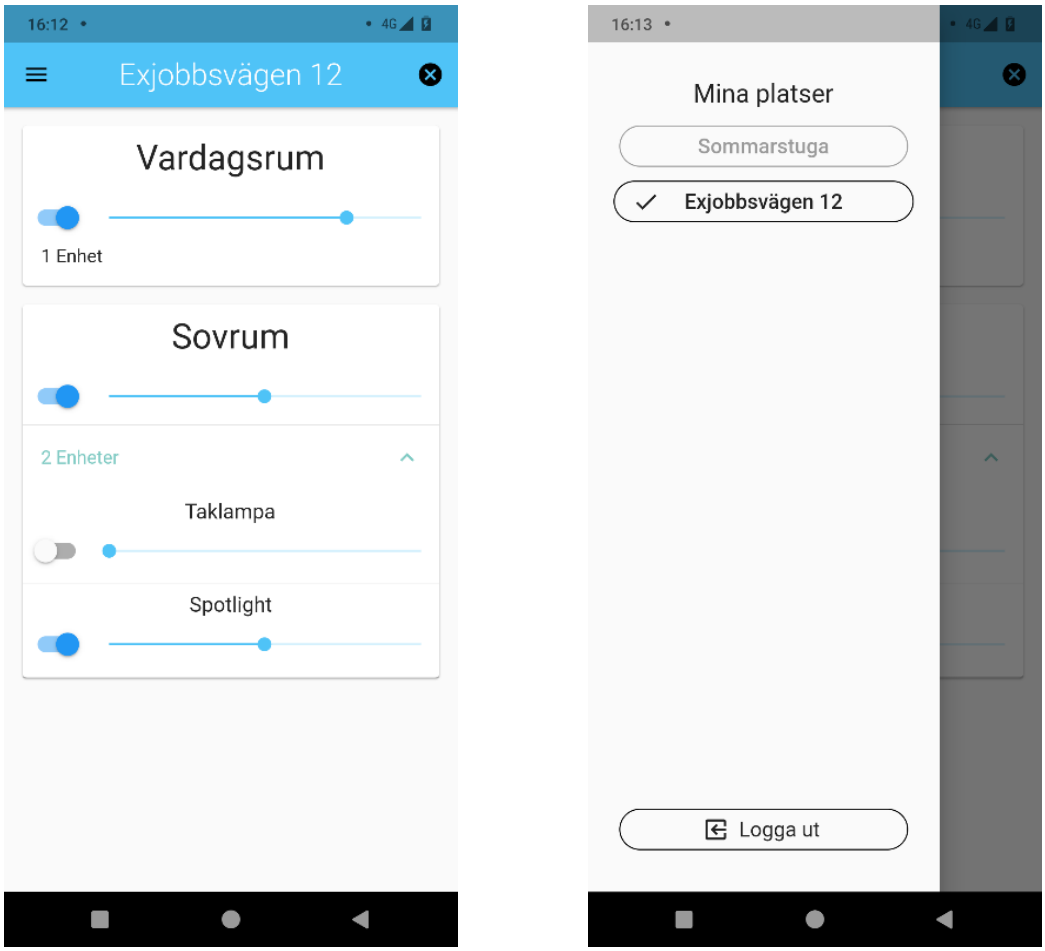
Prototypapplikationen som togs fram uppfyllde alla de krav som ställts i kravspecifikationen. Det finns möjlighet att logga in på befintliga användarkonton för att styra Bluetooth-enheter som är kopplade till sina respektive platser. Styrningen sker på ett sätt som efterliknar det beteende som återfinns i den existerande applikationen, vilket gör applikationerna kompatibla med varandra när det kommer till att vara uppkopplade mot samma plats och styra samma enheter. Det finns också möjlighet att enkelt byta mellan olika platser.

Vid inloggning (se figur 21) kontrolleras email-adressen, så att den följer rätt format och att inget av fälten för email eller lösenord är tomma. Misslyckas inloggningen p.g.a. fel email-adress eller felaktigt lösenord så notifieras användaren om detta.



Figur 21 - Prototypapplikationens inloggningsskärm.

Väl inloggad ansluter applikationen automatiskt mot den första tillgängliga Bluetooth-enhet som tillhör platsen. Efter fullbordad anslutning och autentisering mot Bluetooth-enheten möts användaren av en användarvänlig layout bestående av en kontrollskärm för hela den aktuella platsen (figur 22). Det aktuella platsnamnet återfinns högst upp på skärmen. Under platsnamnet finns det ”kort” för varje rum som tillhör platsen. Samtliga av dessa kort har en uppsättning rumskontroller (bestående av en av-/påknapp och ett reglage för kontroll av ljusstyrka). Med dessa kontroller kan användaren styra alla rummets lampor samtidigt. Finns det mer än en lampa i rummet finns också möjlighet att expandera kortet för att visa individuella kontroller för respektive lampa i rummet. Justeras hela rummets ljusstyrka via rumsreglaget så följer alla rummets individuella reglage rumsreglaget. Slås en lampa av och sedan på igen återgår lampan till sin senast satta ljusstyrka, oavsett om lampan stängts av från en annan telefon än den som slår på lampan igen.



Figur 22 - Kontrollskärmen för en ansluten plats (t.v.). Här visas två kort, ett för varje rum, varav det ena rummet har två lampor. Till höger visas menyn där det finns möjlighet att byta mellan olika platser att ansluta till.

I övre vänstra hörnet (se figur 22) på kontrollskärmen återfinns den menyknapp som öppnar menyn som kontrollerar vilken plats man är ansluten till. Under samma meny finns även valet att logga ut från det aktuella kontot. Loggar användaren ut stängs anslutningen till den aktuella Bluetooth-enheten och den data som sparats om användarens konto raderas från applikationen. Användaren blir också skickad direkt till inloggningsskärmen och måste åter logga in för att kunna ansluta mot en Bluetooth-enhet och styra en plats.

Sammanfattningsvis har det inte varit några större bekymmer att implementera den funktionalitet som företagets system kräver av applikationen. För både Bluetooth- och backend-integrationen fanns det tillgängliga tredjepartsbibliotek som mötte de grundläggande krav som ställts på applikationens funktionalitet. I de fall där man antingen inte vill eller kan integrera tredjepartsbibliotek finns alltid möjligheten i Flutter att på egen hand använda operativsystemets underliggande logik för hantering av t.ex. Bluetooth genom s.k. *platform channels*.

7. Diskussion

7.1 Testresultat

Som förväntat presterade native-applikationerna bäst i sina respektive operativsystem. Båda dessa applikationer på respektive plattform hade kortast uppstartstid och samtidigt minst installationsstorlek. På Android-plattformen var skillnaderna mellan Android-applikationen och cross-platform-ramverken betydligt större, vilket antagligen är en effekt av att Android-plattformen har en större splittring av både versioner och även hårdvara som ramverken ska fungera mot. Detta gör det svårare att optimera ramverken jämfört med IOS där kombinationerna av operativsystemsversioner och hårdvara är mycket färre och därigenom underlättar för utvecklarna att optimera sina ramverk.

Den mest påtagliga skillnaden var installationsstorleken hos respektive operativsystems native-applikation jämfört med cross-platform-applikationerna. I fallet med Xamarin.Forms på Android blev installationsstorlekarna för applikationerna väldigt stora med tanke på hur lite applikationerna innehöll.

Vid mätning av uppstartstider observerades att Android-plattformen generellt har en marginellt större spridning av mätdata inom samma testserie jämfört med IOS. Det ska dock understrykas att denna minimala ökning inte är något som påverkar de presenterade genomsnittliga uppstartstidernas riktighet jämfört med om median skulle använts som lägesmått istället för medelvärde.

Något som avvek från det förväntade resultatet var att ”Bild”-applikationerna genomgående hade snabbare uppstartstid än ”Hello World”-applikationerna. Detta fenomen beror främst på att bilden som användes i applikationen endast är 10 kB stor och att inga beräkningar utförs av applikationen för att skala ned eller upp den. Detta visade sig vara en mindre kostsam operation än att rita ut vektorgrafiken för texten i ”Hello World”-applikationerna.

I IOS presterade alla ramverk förvånansvärt jämnt, speciellt när det kommer till uppstartstid av applikationerna. Efter att ha genomfört tester på IOS-applikationen samt Flutter- och Xamarin.Forms-applikationerna fattades ett beslut att inte inkludera Xamarin.IOS-applikationer i testerna. Beslutet fattades då ingen märkbar skillnad i uppstartstid observerades mellan de olika

ramverken och applikationstyperna, vilket ansågs vara ett tecken på att ytterligare tester inte skulle tillföra något till slutresultatet.

Generellt sett är det svårt att se någon direkt fördel med Xamarin.Forms när det kommer till ren prestanda. De testapplikationer som tagits fram är tänkta att reflektera sådana operationer som ofta sker i mobila applikationer, det är dock svårt att säga om testapplikationerna ger en rättvis bild av hur ramverken presterar i verkliga scenarion. Testerna är endast till för att belysa de prestandamässiga fördelar och/eller nackdelar som kommer som en direkt följd av valet av ramverk.

7.2 Testernas genomförande

För att kunna utföra tester av uppstartstid som är jämförbara både mellan applikationer skrivna i olika ramverk och även över plattformsgränserna fanns det ett behov av att hitta en eller flera mätmetoder som kan användas oavsett plattform och ramverk. Flera lösningar utforskades och testades, däribland att analysera ”displayed”-värdet hos Android och även att analysera ”pre-main”-värdet på IOS-plattformen. Tyvärr visade det sig i slutändan att dessa två metoder inte var lämpliga då Xamarin.Forms inte har möjlighet att utläsa tiden från uppstart till att main-metoden körs på IOS-enheter och ”displayed”-värdet på Android snabbt blir missvisande när man jämför resultat från olika applikationstyper och ramverk.

Till slut föll valet på videoanalys av uppstartstiden då det ger en rättvis bild av vad användaren upplever vid uppstart av applikationen. Fördelen med denna mätmetod är att den är helt plattformsoberoende och inte påverkar prestandan på den enhet som testen utförs på. En nackdel med mätmetoden är att det är tidsödande att analysera videomaterial bildruta för bildruta, vilket leder till att både antalet testenheter och antalet tester per testenhet måste begränsas. Resultaten som observerades från testförsöken hade minimala avvikelser oavsett test, därför begränsades urvalsstorleken till tio testförsök per applikation och enhet.

Efter att ha undersökt möjligheterna att direkt på enheterna spela in det som sker på skärmen så fattades beslutet att istället spela in med en kamera som riktas mot enhetens skärm. Detta beslut togs först och främst då det visade sig svårt att garantera en pålitlig bildfrekvens vid inspelningen. I flera fall var det problematiskt att fånga video med mer än 30 bildrutor/sekund, vilket utgör en källa för oprecisa mätningar och något som inte lämpar sig för analys av individuella bildrutor. Vid inspelning med 30 bildrutor/sekund motsvarar varje bildruta 33,33

millisekunder av video, vilket gör att det kan vara svårt att ta fram tillräckligt precisa mätningar från sådant videomaterial.

Något som också kan ha en inverkan på mätresultaten är de processer och tjänster som körs i bakgrunden på respektive testenhet. För att skapa en så rättvis testmiljö som möjligt har alla applikationer som ligger i RAM-minnet stängts av. Detta för att bara den aktuella testapplikationen skall vara inladdad i RAM-minnet. Även om det för tillfället bara finns en applikation som aktivt körs, så finns det många tjänster och processer som samtidigt körs i bakgrunden på enheten. Utan att ha full tillgång till operativsystemets underliggande tjänster är det svårt att försäkra sig om att inga bakgrundsprocesser eller tjänster tar upp processorkraft.

7.3 Konstruktion av prototypapplikation

Även fast Flutter fortfarande är relativt nytt påträffades inga betydande buggar eller problem relaterade till varken Flutter, Dart eller de tredjepartsbibliotek som använts under projektets gång. De största problem som upplevts under projektet har orsakats av skillnader i hur Bluetooth hanteras av IOS respektive Android, något som inte påverkas av vilket ramverk som används för applikationen.

Problemen som uppenbarat sig under utvecklingsfasen är knutna till hur IOS lagrar Bluetooth-enheternas egenskaper även efter att de kopplats från. IOS sparar så mycket information som möjligt om de Bluetooth-enheter som varit uppkopplade mot telefonen, detta skapar problem då applikationen försöker läsa in data från characteristics som tidigare lagrats men som inte längre är aktuella vid nästa uppkoppling. På samma sätt sparas även prenumerationer på characteristics vid nästkommande start av applikationen, något som inte bör göras. Som konsekvens av detta beteende kan inte prototypapplikationen autentisera sig mot Bluetooth-enheten mer än första gången den ansluter, vid nästkommande försök misslyckas autentiseringen.

För att råda bot på problemet ovan måste BLE-enheterna konfigureras så att båda de fördefinierade ”Generic Access Service” och ”Generic Attribute Service” används, görs inte detta så kommer huvudenheten (telefonen) inte att uppdatera sin sparade data på ett korrekt sätt vid uppkoppling till kringutrustningen. Denna lösning var tyvärr inte möjlig då vi saknade möjlighet att modifiera mjukvaran som körs på företagets hårdvaruprodukter.

Som en alternativ permanent lösning på problemet har prototypapplikationen IOS-specifik kod som bara körs för IOS-plattformen och som ser till att prenumerationer på characteristics bara får startas en gång samt att så mycket som möjligt av den lagrade datan slängs vid nedkoppling från kringutrustningen.

Vissa Android-enheter har också visat sig ha problem att på ett pålitligt sätt ansluta mot kringutrustning via Bluetooth. Detta verkar bero på att Bluetooth-hårdvaran i olika telefoner har drivrutiner som beter sig på olika sätt samt att vissa versioner av Android dras med olika problem relaterade till Bluetooth. I vissa fall kan anslutningen ske på mindre än två sekunder från påbörjad till fullbordad anslutning, i andra fall kan det ta upp emot tio sekunder eller så fullbordas anslutningen inte ö.h.t. Som lösning på detta problem implementerades en återförsöksfunktionalitet där applikationen automatiskt försöker återansluta om anslutningen tappas eller inte kunde fullbordas vid tidigare försök.

En stor fördel med att använda Flutter för att utveckla prototypapplikationen har varit stödet för så kallade hot reloads, vilket underlättat arbetet med både användargränssnitt och applikationens logik. Istället för att kompilera om hela applikationen när en ändring i källkoden måste göras, kan ändringen snabbt göras i utvecklingsmiljön och direkt laddas in i applikationen medan den körs på telefonen. Kompilering samt installation av APK-filer tar normalt ca 30-60 sekunder för en minimal applikation och en modern telefon, detta kan jämföras med en hot reload som tar ca 1 sekund för samma applikation och telefon. Vill man som utvecklare t.ex. utveckla ett användargränssnitt och ha möjlighet att testa sina ändringar efter hand finns det mycket tid att spara genom att inte behöva kompilera om hela applikationens kod gång på gång och istället utföra en hot reload vid varje ny iteration av koden.

8. Samhälls- och miljöpåverkan

Applikationsutveckling med cross-platform-ramverk möjliggör att en mindre grupp utvecklare kan utveckla och underhålla samma applikation till flera plattformar. På detta sätt spenderas mindre resurser, både ekonomiskt och miljömässigt, på att underhålla flera olika kodbaser.

För den enskilde individen som har en bra idé till en applikation, men saknar den kunskap som krävs att utveckla till *både* IOS och Android, öppnas stora möjligheter upp med cross-platform-utveckling. Det räcker då att lära sig bygga en applikation med *ett* ramverk och på så sätt nå ut till i stort sett hela smartphonemarknaden, vilket innebär att cross-platform-teknologier kan vara en demokratiserande kraft inom applikationsutveckling.

9. Slutsats

Det är svårt att se några direkta fördelar med Xamarin.Forms utifrån de prestandaaspekter som undersökts i projektet. Både Xamarin.Android och Flutter presterar genomgående bättre än Xamarin.Forms i alla prestandajämförelser som genomförts på Android. På IOS observerades däremot inga betydande skillnader i prestanda mellan ramverken. Självklart kan det finnas andra fördelar med Xamarin.Forms som ramverk, t.ex. att det är ett väletablerat ramverk med många utvecklare och att en stor variation av tredjepartsbibliotek finns att tillgå, men utifrån ett prestandaperspektiv sett är Xamarin.Forms inte fördelaktigt. Rent prestandamässigt är Flutter det bäst presterande cross-platform-ramverket av de ramverk som jämförts.

Vi har under projektets gång inte upptäckt några tekniska svårigheter som skulle begränsa företaget om de skulle vilja byta ramverk och utveckla sin applikation i Flutter. Ramverket lämpar sig väl för den typ av applikation som företaget utvecklar och vi anser att det skulle finnas många fördelar vid ett eventuellt byte av ramverk, speciellt när det kommer till ren prestanda. Förutom prestandaaspekten skulle ett byte till Flutter underlätta utvecklingsprocessen avsevärt då hot reload kan korta ned tiden när nya funktioner utvecklas i applikationen.

Ett eventuellt byte av ramverk skulle dock innebära mycket arbete i form av rent översättningsarbete från C# till Dart, i de delar som är möjliga att återanvända från den nuvarande applikationen. Alla UI-delar och ramverksspecifika delar skulle dock behöva skrivas om från grunden, vilket är ett betydande arbete. Detta resulterar i att det inte blir lika självklart att Flutter är ett bättre alternativ för företaget.

En annan möjlighet är att företaget skulle kunna byta ramverk till Xamarin.Android/Xamarin.IOS för att specifikt förbättra applikationens uppstartstid på Android-plattformen, detta förutsätter dock att resultaten från prestandajämförelserna också speglar ramverkens prestanda i mer komplexa applikationer. Ett sådant byte skulle innebära att företagets utvecklare behöver utveckla och underhålla två separata användargränssnitt för applikationen men att man i gengäld kan behålla den redan existerande logiken i applikationen.

Referenser

[1] Newzoo, ”Global Mobile Market Report,” 2018. [Online]. Tillgänglig: https://resources.newzoo.com/hubfs/Reports/Newzoo_2018_Global_Mobile_Market_Report_Free.pdf. [Använd 13 03 2019].

[2] App Annie, ”2017 Retrospective,” 2017. [Online]. Tillgänglig: https://www.appannie.com/en/insights/market-data/app-annie-2017-retrospective/. [Använd 20 02 2019].

[3] Statista, ”Global Market Share Held by the Leading Smartphone Operating Systems in Sales to end Users from 1st Quarter 2009 to 2nd Quarter 2018,” 2018. [Online]. Tillgänglig: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. [Använd 20 02 2019].

[4] The Linux Information Project, ”Cross-Platform Definition,” [Online]. Tillgänglig: http://www.linfo.org/cross-platform.html. [Använd 31 01 2019].

[5] P. Grzmil, M. Skublewska-Paszkowska, E. Łukasik och J. Smołka, ”Performance Analysis of Native and Cross-Platform Mobile Applications,” 2017. [Online]. Tillgänglig: https://www.researchgate.net/publication/320039474_performance_analysis_of_native_and_cross-platform_mobile_applications. [Använd 13 03 2019].

[6] IBM, ”The AOT compiler,” IBM, [Online]. Tillgänglig: https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/aot.html. [Använd 07 05 2019].

[7] Infoq, ”Androir to Include Ahead-of-Time Compiler,” 2014. [Online]. Tillgänglig: https://www.infoq.com/news/2014/07/ahead-of-time-compiler-os. [Använd 08 02 2019].

[8] IBM, ”The JIT Compiler,” [Online]. Tillgänglig: https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/jit_overview.html. [Använd 08 02 2019].

[9] C. Lattner, ”LLVM,” [Online]. Tillgänglig: https://www.aosabook.org/en/llvm.html#fig.llvm.lcom. [Använd 08 02 2019].

[10] Guardsquare, ”Proguard,” [Online]. Tillgänglig: https://www.guardsquare.com/en/products/proguard. [Använd 08 02 2019].

[11] Google, ”Shrink Your Code and Resources,” [Online]. Tillgänglig: https://developer.android.com/studio/build/shrink-code. [Använd 09 02 2019].

[12] Bluetooth SIG, ”The right radio, for the right job.,” [Online]. Tillgänglig: https://www.bluetooth.com/bluetooth-technology/radio-versions/. [Använd 08 05 2019].

[13] R. Davidson, Akiba, C. Cufí, K. Townsend, *Getting Started with Bluetooth Low Energy*, Sebastopol, Kalifornien, USA: O'Reilly Media, Inc, 2014.

[14] Karl Torvmark, Texas Instruments, ”Three flavors of Bluetooth®: Which one to choose?,” 03 2014. [Online]. Tillgänglig: http://www.ti.com/lit/wp/swry007/swry007.pdf. [Använd 07 05 2019].

[15] ”Bluetooth mesh networking FAQs,” Bluetooth SIG, [Online]. Tillgänglig: https://www.bluetooth.com/bluetooth-technology/topology-options/le-mesh/mesh-faq/. [Använd 08 05 2019].

[16] ”Topology Options,” Bluetooth SIG, [Online]. Tillgänglig: https://www.bluetooth.com/bluetooth-technology/topology-options/. [Använd 08 05 2019].

[17] Silicon Labs, ”KBA_BT_0406: Adaptive Frequency Hopping,” Silicon Labs, 20 09 2018. [Online]. Tillgänglig: https://www.silabs.com/community/wireless/bluetooth/knowledge-base.entry.html/2018/09/20/adaptive_frequencyh-rIn0. [Använd 08 05 2019].

[18] M. d. Icaza, ”MonoTouch 1.0 Goes Live,” [Online]. Tillgänglig: https://tirania.org/blog/archive/2009/Sep-14.html. [Använd 21 04 2019].

[19] The H, ”Mono for Android brings C# to Android,” 2011. [Online]. Tillgänglig: http://www.h-online.com/open/news/item/Mono-for-Android-brings-C-to-Android-1223483.html. [Använd 15 03 2019].

[20] Microsoft, ”Microsoft to Acquire Xamarin and Empower More Developers to Build Apps on any Device,” 2016. [Online]. Tillgänglig: https://blogs.microsoft.com/blog/2016/02/24/microsoft-to-acquire-xamarin-and-empower-more-developers-to-build-apps-on-any-device/. [Använd 08 02 2019].

[21] Microsoft, ”Building Cross Platform Applications Overview,” 23 03 2017. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/overview. [Använd 12 05 2019].

[22] Microsoft, ”iOS App Architecture,” 2017. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/xamarin/ios/internals/architecture. [Använd 08 02 2019].

[23] Microsoft, ”Architecture,” 2018. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/xamarin/android/internals/architecture. [Använd 08 02 2019].

[24] Microsoft, ”Preparing an Application for Release,” 2018. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/xamarin/android/deploy-test/release-prep/?tabs=windows#aot-compilation. [Använd 14 03 2019].

[25] Xamarin, ”Meet Xamarin.Forms: 3 Native UIs, 1 Shared Code Base,” 2014. [Online]. Tillgänglig: https://blog.xamarin.com/meet-xamarin-forms-3-native-uis-1-shared-code-base/. [Använd 08 02 2019].

[26] Altexsoft, ”Performance Comparison: Xamarin.Forms, Xamarin.iOS, Xamarin.Android vs Android and iOS Native Applications,” 2017. [Online]. Tillgänglig: https://www.altexsoft.com/blog/engineering/performance-comparison-xamarin-forms-xamarin-ios-xamarin-android-vs-android-and-ios-native-applications/. [Använd 14 03 2019].

[27] S. Ladd, ”Announcing Flutter Beta 1: Build Beautiful Native Apps,” 2018. [Online]. Tillgänglig: https://medium.com/flutter-io/announcing-flutter-beta-1-build-beautiful-native-apps-dc142aea74c0. [Använd 15 03 2019].

[28] Google, ”Technical Overview,” [Online]. Tillgänglig: https://flutter.io/docs/resources/technical-overview. [Använd 08 02 2019].

[29] Google, ”Does Flutter use my system’s OEM widgets?,” [Online]. Tillgänglig: https://flutter.dev/docs/resources/faq#does-flutter-use-my-systems-oem-widgets. [Använd 22 03 2019].

[30] Google, ”Introduction to Widgets,” [Online]. Tillgänglig: https://flutter.dev/docs/development/ui/widgets-intro. [Använd 22 03 2019].

[31] Google, ”Hot reload,” [Online]. Tillgänglig: https://flutter.dev/docs/development/tools/hot-reload. [Använd 22 03 2019].

[32] T. Occhino, ”React Native: Bringing modern web techniques to mobile,” 26 03 2015. [Online]. Tillgänglig: https://code.fb.com/android/react-native-bringing-modern-web-techniques-to-mobile/. [Använd 22 03 2019].

[33] React Native, ”React Native Internals,” [Online]. Tillgänglig: https://www.reactnative.guide/3-react-native-internals/3.1-react-native-internals.html. [Använd 22 03 2019].

[34] Facebook, ”Introducing Hot Reloading,” 2016. [Online]. Tillgänglig: https://facebook.github.io/react-native/blog/2016/03/24/introducing-hot-reloading. [Använd 08 02 2019].

[35] Facebook, ”JavaScript Environment,” [Online]. Tillgänglig: https://facebook.github.io/react-native/docs/javascript-environment. [Använd 30 04 2019].

[36] A. Bradley, ”Ionic Framework Beta Released!,” 26 03 2014. [Online]. Tillgänglig: https://blog.ionicframework.com/ionic-framework-beta-released/. [Använd 22 03 2019].

[37] The Appache Software Foundation, ”Appache Cordova,” [Online]. Tillgänglig: https://cordova.apache.org/. [Använd 08 02 2019].

[38] M. Lynch, ”Introducing Ionic 4: Ionic for Everyone,” 23 01 2019. [Online]. Tillgänglig: https://blog.ionicframework.com/introducing-ionic-4-ionic-for-everyone/. [Använd 22 03 2019].

[39] Mozilla, ”Web Components,” [Online]. Tillgänglig: https://developer.mozilla.org/en-US/docs/Web/Web_Components. [Använd 22 03 2019].

[40] Ionic, ”WebView,” [Online]. Tillgänglig: https://ionicframework.com/docs/building/webview. [Använd 30 04 2019].

[41] Google, ”Application Fundamentals,” [Online]. Tillgänglig: https://developer.android.com/guide/components/fundamentals.html. [Använd 29 04 2019].

[42] Google, ”Implementing ART Just-In-Time (JIT) Compiler,” [Online]. Tillgänglig: https://source.android.com/devices/tech/dalvik/jit-compiler.html. [Använd 29 04 2019].

[43] Google, ”Build and run your app,” [Online]. Tillgänglig: https://developer.android.com/studio/run#instant-run. [Använd 29 04 2019].

[44] Apple, ”Swift,” [Online]. Tillgänglig: https://developer.apple.com/swift/. [Använd 29 04 2019].

[45] Microsoft, ”iOS App Architecture,” [Online]. Tillgänglig: https://docs.microsoft.com/en-us/xamarin/ios/internals/architecture#aot. [Använd 29 04 2019].

[46] Google, ”App startup time,” [Online]. Tillgänglig: https://developer.android.com/topic/performance/vitals/launch-time#time-initial. [Använd 03 05 2019].

[47] Google, "App startup time," [Online]. Tillgänglig: https://developer.android.com/topic/performance/vitals/launch-time#cold. [Använd 03 05 2019].

[48] P. DeMarco, ”FlutterBlue,” 03 04 2018. [Online]. Tillgänglig: https://github.com/pauldemarco/flutter_blue. [Använd 06 05 2019].