Ida Rask

IPHONEN SOVELTUVUUS REAALIAI-KAISEEN KIIHTYVYYSANTURIN DA-TAN KERÄÄMISEEN

Opinnäytetyö

Tekniikan ammattikorkeakoulututkinto

Peliohjelmoinnin koulutus

2024

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Insinööri (AMK)
Tekijä/Tekijät	Ida Rask
Työn nimi	iPhonen soveltuvuus reaaliaikaiseen kiihtyvyysanturin datan keräämiseen
Toimeksiantaja	Gleap Health Technologies Oy
Vuosi	2024
Sivut	39 sivua
Työn ohjaaja(t)	Pekka Vilpponen

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli tutkia iPhonen soveltuvuutta reaaliaikaiseen kiihtyvyysanturin datan keräämiseen kehittämällä toimeksiantajan käyttöön kaksi versiota puhelimen omaa kiihtyvyysanturia taustalla nauhoittavasta mobiilisovelluksesta. iOS on käyttöjärjestelmänä hyvinkin itsenäinen ja suunniteltu optimoimaan resurssien käyttöä mahdollisimman hyvän käyttäjäkokemuksen varmistamiseksi. Lisäksi sovelluksen aktiivisuustila vaikuttaa siihen, mitä milläkin hetkellä voidaan tehdä. Toimeksiantaja halusikin siis selvittää, onko iPhonen oman kiihtyvyysanturin nauhoittaminen taustalla ylipäätään mahdollista ja mitä kaikkea pitäisi ottaa huomioon kaupallisen sovelluksen kehittämisessä.

Tutkimusongelmaa lähestyttiin konstruktiivisen tutkimuksen kautta. Oikeastaan ainut keino selvittää, onko iPhonen kiihtyvyysanturin nauhoittaminen taustalla mahdollista, oli kehittää sovellus, jolla testata tätä. Koska konstruktiivinen tutkimus kytkeytyy hyvin vahvasti jo olemassa olevaan teoriaan aiheesta, aloitettiin tutkimus tutustumalla Applen omaan dokumentaatioon siitä, miten iOS käyttöjärjestelmänä toimii, miten sovelluksen aktiivisuustila vaikuttaa sen toimintaan, millaisia taustasuoritustiloja sovellukseen on mahdollista implementoida, millaisia vaihtoehtoja kiihtyvyysanturin nauhoittamiselle on ja mitä sovelluskehyksiä tarvitaan haluttujen toiminnallisuuksien implementoimiseen. Lisäksi tutustuttiin aiempiin tutkimuksiin iPhonen kiihtyvyysanturin tarjoaman datan laadusta, jotta varmistuttiin siitä, että kerättävä data on varmasti luotettavaa.

Haluttujen toiminnallisuuksien kehittäminen aloitettiin pienistä prototyypeistä käyttäen teoriapohjaa hyödyksi. Lopullinen konstruktio toteutettiin osaksi jo olemassa olevaa monialustaiseksi suunniteltua mobiilisovellusta, jonka Android-puoli oli jo kehityksessä. Pääasiallisena työkaluna oli Xcode ja natiivipuolen toiminnallisuuksien implementoimiseen käytettiin Swift-ohjelmointikieltä. Sovellusversioiden toimivuutta testattiin vapaaehtoisista koostuvan testiryhmän avulla.

Molemmat sovellusversiot saatiin toteutettua, mutta ei täysin toimeksiantajan kriteerien mukaisesti. Ongelmia aiheutti käyttöjärjestelmän suorittama optimointi, eivätkä sovellukset pysyneet auki taustalla halutulla tavalla. Tutkimusongelmaa ei siis saatu täysin ratkaistua tässä työssä käytettyjen metodien avulla, ja haluttujen toiminnallisuuksien toteuttaminen vaatii vielä lisää työtä.

Asiasanat: mobiilisovellukset, iPhone, iOS, ohjelmistokehitys

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author (authors)	Ida Rask
Thesis title	iPhone’s suitability for collecting real-time accelerometer data
Commissioned by	Gleap Health Technologies Oy
Time	2024
Pages	39 pages
Supervisor	Pekka Vilpponen

ABSTRACT

The objective of this thesis was to study iPhone’s suitability for collecting real-time accelerometer data by developing two versions of a mobile application that records iPhone’s internal accelerometer in the background to the commissioner. iOS, being a highly autonomous operating system, is designed to optimize resource usage for ensuring the best possible user experience. Additionally, the activity state of the application determines what actions can be performed at any given moment. Therefore the commissioner wanted to determine whether recording iPhone's internal accelerometer in the background is possible and what should be considered when developing a commercial application.

The research problem was approached through constructive research methodology. The only way to find out if recording iPhone's accelerometer in the background is possible, was to develop an application to test this. As constructive research is closely aligned with already existing theory on the subject, the research began by examinating Apple's documentation on how iOS works as an operating system, how the activity state of an application affects its operation, what background execution modes can be used, what kind of options are there for recording the accelerometer, and what frameworks are needed for implementing the desired functionalities. Furthermore, prior studies on the quality of the accelerometer data provided by iPhone were reviewed to ensure the reliability of collected data.

The development of the desired functionalities started with the creation of small prototypes, with the help of the theory base. The final construction was implemented into an existing multiplatform mobile application, where the Android counterpart was already in development. Xcode served as the primary tool, with Swift programming language used for implementing the native side functionalities. The functionality of the application versions was evaluated through testing, using a test group consisting of volunteers.

While both application versions were more or less successfully developed, they did not entirely meet the commissioner’s criteria. Challenges arose due to the operating system's optimization, resulting in applications not remaining active in the background as intended. Consequently, the research problem was not fully resolved using the employed methodologies, indicating the need for further work to implement the desired functionalities.

Keywords: mobile applications, iPhone, iOS, software development

SISÄLLYS

1 JOHDANTO 5
2 TUTKIMUSASETELMA 6
2.1 Tutkimusongelma 6
2.2 Tutkimuskysymykset 6
2.3 Tutkimusote 7
3 TEOREETTINEN VIITEKEHYS 8
3.1 GPS ja kiihtyvyysanturi 8
3.2 Työkalut 11
3.3 iOS-kehitys 12
4 ENSIMMÄINEN SOVELLUSVERSIO 15
4.1 Kehitys 16
4.2 Testaus 26
5 TOINEN SOVELLUSVERSIO 28
5.1 Kehitys 28
5.2 Testaus 30
6 TULOKSET 32
7 JOHTOPÄÄTÖKSET 33
8 POHDINTA 35
LÄHTEET 37

1 JOHDANTO

Tässä opinnäytetyössä tutkitaan, soveltuuko iPhone reaaliaikaiseen kiihtyvyysanturin datan keräämiseen. Applella on oma sovelluskehys, joka tarjoaa useita keinoja saada dataa kiihtyvyysanturista (Getting raw… s.a.; recordAccelerometer… s.a.) sekä erilaisia taustasuoritustiloja (Configuring… s.a.) ja taustatehtäviä (Background Tasks s.a.), joilla voi hallita sovelluksen käyttäytymistä silloin, kun se ei ole avattuna. Applen dokumentaatio on kuitenkin vaikeasti ymmärrettävissä ja suurimmaksi osin siitä puuttuvat kokonaan esimerkit, miten eri sovelluskehyksiä käytännössä hyödynnetään. Työn aikana tutustutaan siis erilaisiin työkaluihin, joilla iOS-kehitystä voi tehdä, iOS-mobiilisovelluksen erilaisiin tiloihin sekä Applen tarjoamiin sovelluskehyksiin ja hyödynnetään niitä käytännössä.

Opinnäytetyön aihe tuli toimeksiantajalta, joka on vuonna 2016 perustettu haminalainen yritys Gleap Health Technologies Oy. Gleap Health Technologies on osa vuonna 2015 perustettua Arctic Nutrition Oy:tä ja se keskittyy ohjelmistojen suunnitteluun ja valmistukseen molempien yritysten tarpeeseen. Yrityksellä on kehitteillä terveyden seurantaan mobiilisovellus, joka hyödyntää kiihtyvyysanturista saatavaa dataa ja ulkoisen anturin lisäksi halutaan hyödyntää puhelimen omaa anturia, jos käyttäjä ei halua panostaa ulkoisen anturin hankintaan.

Opinnäytetyön tavoitteena on tuottaa toimeksiantajalle kiihtyvyysanturia nauhoittavasta iPhone-sovelluksesta kaksi versiota, joilla testata haluttuja ominaisuuksia ennen varsinaisen kaupallisen tuotteen toteuttamista. Ensimmäisen version tulisi kerätä kiihtyvyysdataa ympäri vuorokauden, jotta voidaan tarkastella sitä, kykeneekö iPhone tallentamaan dataa katkeamatta vai puuttuuko käyttöjärjestelmä asiaan, jos sovellus kuluttaa liikaa resursseja. Toisen version tulisi olla lähempänä valmiin tuotteen vaatimuksia, ja sen tulisi nauhoittaa kiihtyvyysanturin dataa vain silloin, kun käyttäjä liikkuu, jotta saadaan minimoitua turhan datan määrä, mutta myös optimoitua sovelluksen resurssien käyttö. Koska toimeksiantajan suunnittelema sovellus on tarkoitus julkaista niin

iPhonelle kuin Android-puhelimillekin, oli monialustaisen testisovelluksen kehittäminen jo aloitettu, joten iPhone-implementaatio toteutetaan osaksi tätä olemassa olevaa sovellusta.

2 TUTKIMUSASETELMA

Tässä luvussa käsitellään tutkimusasetelmaa tutkimusongelman, -kysymysten ja -otteen kautta.

2.1 Tutkimusongelma

Mahdollisimman hyvän käyttäjäkokemuksen toteutumiseksi Apple haluaa, että iPhonessa on mahdollisimman hyvä akunkesto, joten iOS on käyttöjärjestelmänä suunniteltu hyvinkin tarkasti optimoimaan kaikkien resurssien käyttö. Puhelin oppii, milloin käyttäjä käyttää tiettyjä sovelluksia, ja sen perusteella vapauttaa resursseja sovellusten käyttöön ja toisaalta myös evää resursseja, jos näyttää siltä, että joku sovellus esimerkiksi vie liikaa akkua. Vain muutaman tyyppiset sovellukset, kuten mm. kartta- ja musiikintoistosovellukset, voivat pyytää lupaa erillisille taustasuoritustiloille, jotka antavat sovellukselle luvan toimia taustalla jatkuvasti. Sovelluskehittäjällä on siis hyvin vähän keinoja taata säännöllisiä (ainakaan niin säännöllisiä, kuin itse haluaisi) taustapäivityksiä sovellukseen. (Apple Developer Videos 2020.) Myös se, missä aktiivisuustilassa sovellus milläkin hetkellä on, vaikuttaa siihen, mitä taustalla voi tapahtua. (Managing… s.a.)

Vaikuttaa siis siltä, että iOS on käyttöjärjestelmänä haastava ympäristö toteuttaa sovellus, joka pystyisi jatkuvasti nauhoittamaan puhelimen kiihtyvyysanturia reaaliaikaisesti myös taustalla ollessaan. Tarkoituksena onkin tutkia, soveltuuko iPhone ylipäätään tällaisen sovelluksen kehittämiseen suunnitelluilla ominaisuuksilla vai täytyykö jostain joustaa, jotta saadaan edes jotenkin samantyyppisesti toimiva ratkaisu toteutettua.

2.2 Tutkimuskysymykset

Kuten tutkimusongelmasta voi huomata, moni asia vaikuttaa toisiinsa, ja tästä voidaan johtaa seuraavanlaisia tutkimuskysymyksiä:

1. Kuinka mobiilisovelluksen aktiivisuustila vaikuttaa sen toimintaan?
2. Mitä eri taustasuoritustiloja Apple tarjoaa ja miten ne eroavat toisistaan?
3. Mitä keinoja Apple tarjoaa iPhonen kiihtyvyysanturin nauhoittamiseksi?

Applella on laaja kirjo erilaisia sovelluskehyksiä ja taustasuoritustiloja, eikä niitä kaikkia voida käsitellä tämän työn rajoissa, joten tarkastelun alle otettiin vain sellaiset olennaiset osat, joista oli hyötyä juuri tämän sovelluksen toteuttamisessa.

2.3 Tutkimusote

Poiketen perinteisistä tutkimuksista, joiden tavoitteena on kuvata, selittää ja ymmärtää ilmiöitä, interventiotutkimus pyrkii muutokseen tekemällä asiat toisin tai keksimällä paremman ratkaisun. (Kananen 2017, 10.) Konstruktiivinen tutkimus on yksi interventiotutkimuksen muodoista, ja se pyrkii toteuttamaan muutoksen nimensä mukaisesti konstruktiolla eli ratkaisemaan ongelman mallin, kuvion, suunnitelman, organisaation, koneen tai vastaavan rakentamisen avulla. Jotta tutkimus on konstruktiivinen, tutkija on itse mukana muutoksessa, sen on kytkeydyttävä aikaisempaan teoriaan, kirjallisuuteen ja tutkimukseen aiheesta sekä perustuttava uutuuteen ja sen toimivuus on todettava käytännössä. Käytännön toimivuus testataan pilotoinnilla. (Kananen 2017, 14–16.) Kuvasta 1 nähdään konstruktiivisen tutkimuksen osat.

Kuva 1. Konstruktiivisen tutkimuksen osat (mukaillen Kananen 2017, 23)

Ongelman ratkaisemiseksi pyritään toteuttamaan uusi sovellus. Sovelluksen toteutus kytkeytyy hyvin vahvasti olemassa olevaan dokumentaatioon Applen sovelluskehyksistä ja niiden käytöstä (yhteys teoriaan), kun mietitään, miten halutut ominaisuudet saadaan toteutettua. Lopullisen kaupallisen sovelluksen – jota varten tutkimustyötä ja testausta tehdään – idea on uusi (teoreettinen merkitys), eikä markkinoilta löydy vielä ainuttakaan samankaltaista sovellusta.

Sovelluksen toimivuutta testataan testiryhmän kautta (käytännön toimivuus) ja testeistä saatua dataa analysoidaan sen luotettavuuden todentamiseksi (pilotointi). Kaikki kuvassa 1 esitellyt osat ovat siis tässä tutkimuksessa läsnä ja voidaan sanoa tutkimuksen olevan konstruktiivinen.

3 TEOREETTINEN VIITEKEHYS

Tässä luvussa avataan tutkimuksen kannalta oleellisia termejä, tutustutaan aikaisempiin tutkimuksiin sekä käydään läpi tärkeimpiä työkaluja ja iOS-kehitykseen liittyviä kokonaisuuksia.

3.1 GPS ja kiihtyvyysanturi

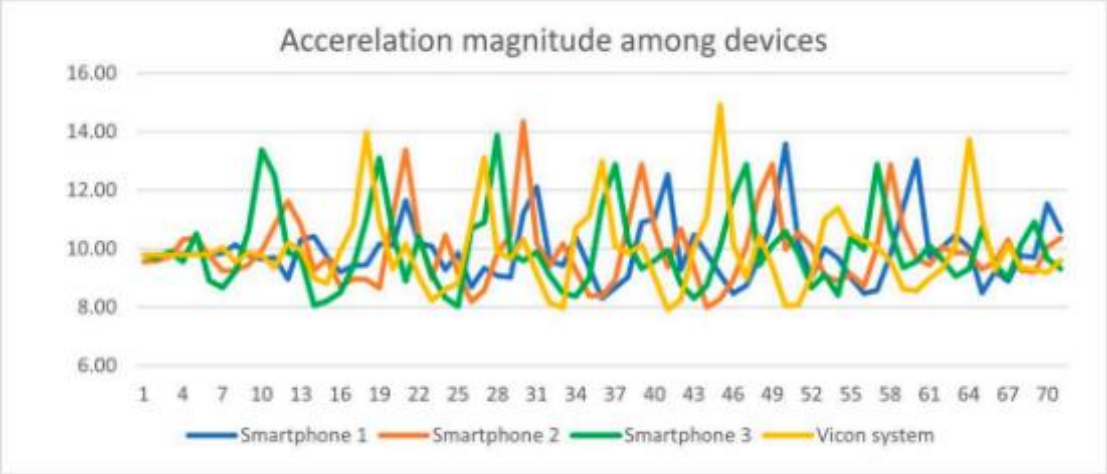
GPS (*global positioning system*) koordinaatit ovat yksilöllisiä tunnistetietoja tarkoille maantieteellisille sijainneille maapallolla, jotka ilmaistaan leveys- ja pituusasteen yhdistelmänä joko aakkosnumeerisena esim. N40° 44,9064', W073° 59,0735' tai puhtaasti numeerisena esim. 40,748440, -73,984559. GPS-paikannus toimii 24 GPS-satelliitin kautta, jotka kiertävät maapalloa. Sijaintitiedot välitetään radiosignaaleilla maahan, jossa laitteet ottavat vastaan sijaintitietoja kolmelta tai useammalta satelliitilta. GPS-tiedot ovat käytettävissä kaikissa laitteissa, jotka voivat lähettää ja vastaanottaa niitä. (Kirvan 2022.)

Kiihtyvyysanturi puolestaan on laite, joka mittaa kohteen nopeuden muutosnopeutta eli nimensä mukaisesti kiihtyvyyttä. Kiihtyvyyttä voidaan mitata joko metreinä sekunnissa per neliö (m/s2) tai G-voimina, jolloin yksi G vastaa noin 9,8 m/s2. Kiihtyvyysanturilla voidaan havaita joko staattisia, kuten painovoima, tai dynaamisia voimia, kuten tärinä tai liike. Kiihtyvyyttä voidaan mitata yhdellä, kahdella tai kolmella akselilla. (Sparkfun s.a.)

Älypuhelimissa sisäänrakennettu kiihtyvyysanturi mittaa laitteen lineaarista kiihtyvyyttä, jolla voidaan seurata mm. ravistamista, kallistusta, heilumista ja pyörimistä tai esimerkiksi muuttaa sovelluksen orientaatiota. Älypuhelimessa kiihtyvyyttä mitataan yleensä kolmella akselilla (X, Y, Z). Apple on integroinut kiihtyvyysanturin kaikkiin iPhoneihin neljännestä sukupolvesta eteenpäin. Koska kiihtyvyysanturilla voidaan seurata käyttäjän liikkumista, se on laajalti käytössä terveys- ja urheilusovelluksissa. (Sharma 2020.)

Manohar ym. (2011) tutkivat iPhonen kiihtyvyysanturin datan luotettavuutta vertaamalla sitä fyysisen aktiivisuudenvalvontajärjestelmäpuvun (engl. *Physical Activity Monitoring System, PAMS*) – joka toimi tutkimuksen kultaisena standardina – antamaan dataan. Dataa kerättiin 13 akselilla (10 akselia PAMS:sta ja 3 akselia iPhonen kiihtyvyysanturista) ja tietorivejä kertyi 210 000, mikä edusti 2 730 000 liikettä. Tutkimuksessa havaittiin puhelimen kiihtyvyysanturin kykenevän erottamaan kiihtyvyyden muutokset luotettavasti jopa 0,8 km/h lisäyksillä. Tätä verrattiin PAMS:in mittaamaan dataan ja havaittiin puhelimen kiihtyvyysanturin datan olevan oikeanlaista ja tarkkaa verratessa PAMS:iin.

Grouios ym. (2022) puolestaan tutkivat eri puhelinmallien kiihtyvyysanturien antamaa dataa ja vertasivat sitä yleisesti hyväksyttyyn kultaiseen malliin, Vicon MX liikkeenkaappausjärjestelmään, joka on tunnettu nykyajan yhtenä edistyneimmistä digitaalisista optisista liikkeenkaappausjärjestelmistä. Tutkimuksessa käytettiin kolmea sillä hetkellä käytetyintä brändiä ja laitesukupolvea: iPhone 12 Pro Max, Samsung Galaxy S21 ja Huawei P Smart. Tarkoituksena oli selvittää, kuinka paljon nykyaikaisten puhelinten tuottama kiihtyvyysdata eroaa tästä Vicon MX kultaisesta standardista. Testattujen puhelinten havaittiin olevan kelvollisia ja luotettavia laitteita lineaaristen kiihtyvyyksien arvioimiseen, eikä kultaiseen standardiin verrattaessa kiihtyvyysdatan välillä ollut merkittäviä eroja datan tarkkuudessa (kuva 2).



Kuva 2. Kiihtyvyyksistä laskettu magnitudi eri puhelinmerkkien ja Vicon MX:n välillä (Grouios ym. 2022)

Opinnäytetyön osalta kiinnostus on ainoastaan kiihtyvyyksistä laskettavasta magnitudista, ja kuten kuvasta 2 voidaan päätellä, eri puhelinmallien välillä ei ole huomattavia eroja kultaiseen standardiin. Näiden aiempien tutkimusten perusteella voidaan päätellä, että iPhonesta saatu anturidata on luotettavaa.

Kuten aiemmin on mainittu, kiihtyvyysanturia käytetään laajalti terveys- ja urheilusovelluksissa. Esimerkiksi App Storesta saatavista suosituista urheilusovelluksista Sports Tracker (Sports Tracker… s.a.), Strava (Strava… s.a.), Suunto (Suunto s.a.) ja Polar Beat (Polar Beat… s.a.) ilmoittavat keräävänsä terveys- ja kuntoilutietoja, joihin myös kiihtyvyysanturin luvat kuuluvat, joten tästä voisi päätellä niiden käyttävän myös anturia sovelluksissaan, vaikka täysin varmaa tietoa tästä ei olekaan.

Näiden lisäksi App Storessa on myös muutamia muita sovelluksia, jotka hyödyntävät puhelimen kiihtyvyysanturia. Physics Toolbox Accelerometer (Physics… s.a.), Acc.elerometer (Acc.elerometer s.a.) ja Anahertz – Frequency Analysis (Anahertz… s.a.) pystyvät kaikki keräämään kiihtyvyysanturista tietoja, ja tiedot näytetään käyttäjälle erilaisina graafeina. Speedometer∞ (Speedometer∞ s.a.) puolestaan käyttää kiihtyvyysanturia laskemaan senhetkistä nopeutta, ja Accelmeter (AccelMeter s.a.) taas visualisoi kiihtyvyyksien suunnat ja voiman nuolina ja numeroina.

Edellä mainitut urheilusovellukset käyttävät kaikki myös käyttäjän sijaintitietoja, mikä antaa lisää mahdollisuuksia sovellukselle toimia taustalla, esimerkiksi silloin, jos puhelimen näyttö on sammutettuna ja laite käyttäjän taskussa lenkillä. Jälkimmäiset, kiihtyvyysanturia nauhoittavat sovellukset puolestaan luottavat siihen, että sovellus on koko ajan auki käyttäjän näytöllä, jotta dataa voidaan näyttää reaaliajassa. Yksikään näistä ei myöskään ole puhtaasti terveyssovellus, joten voidaan olettaa, ettei markkinoilla ole vielä yhtään samanlaista sovellusta. Toimeksiantajan sovellukseen ei ole kuitenkaan suunniteltu sijaintitietojen käyttöä, mutta näiden sovellusten pohjalta on vakaasti harkittava myös niiden keräämistä, jos se on ainut keino saada sovellus toimimaan suunnitellusti.

3.2 Työkalut

Flutter on Googlen kehittämä avoimen lähdekoodin käyttöliittymätyökalu tai rajapinta, jolla on mahdollista kehittää monialustaisia sovelluksia käyttäen vain yhtä koodikantaa. Sama koodi kääntyy siis natiivisti jokaiselle tuetulle alustalle, eikä niille näin ollen tarvitse tehdä eri koodeja. Flutter tukee mobiili-, web- ja työpöytäsovellusten kehittämistä (kuva 3) macOS-, ChromeOS-, Linux- ja Windows-tietokoneilla. Flutter itsessään on rakennettu käyttäen C-, C++- ja Dart-ohjelmointikieliä ja Flutter-rajapinnalla sovellusten kehittäminen tapahtuu näistä viimeisellä. (FAQ s.a.)

As of the current stable release, support for deploying Flutter apps is shown in the following table:

Platform version	Supported	Best effort	Unsupported
Android SDK	21-34	19-20	18-
iOS	16	12-15, 17	11-, arm7v 32-bit
Linux Debian	10-12	9-	any 32-bit
Linux Ubuntu	20.04 LTS	20.10-23.04	any 32-bit
macOS	Ventura (13)	Mojave (10.14) to Monterey (12), Sonoma (14)	High Sierra (10.13-)
web - Chrome	latest 2 releases	96+	
web - Firefox	latest 2 releases	99+	
web - Safari	latest 2 releases	14+	
web - Edge	latest 2 releases	96+	
Windows	10	7, 8, and 11	Vista-, any 32-bit

Kuva 3. Lista alustoista, joille kehittämistä Flutter tukee (Supported deployment platforms s.a.)

Dart on Googlen kehittämä avoimen lähdekoodin ohjelmointikieli, johon pääsi ensimmäistä kertaa tutustumaan lokakuussa 2011. Ensimmäinen tuotantoversio julkaistiin marraskuussa 2013. Dart yhdistelee parhaita puolia useista eri ohjelmointikielistä, kuten Java, C#, JavaScript, Python ja Ruby. (Ridjanovic & Balbaert 2013, 9–10.)

Flutter ja sen käyttämä Dart-ohjelmointikieli ovat hyvin pienessä osassa tässä opinnäytetyössä, sillä käytännössä koko Flutter-puoli oli jo ohjelmoitu pieniä käyttöjärjestelmäkohtaisia muokkauksia lukuun ottamatta. Vaikka tämä työ

keskittyykin vain iOS-puolen toiminnallisuuksiin, on isommassa skaalassa tarkoituksena tuottaa sovellus, joka toimii niin Android- kuin iOS-laitteillakin. Näin ollen koen tärkeäksi esitellä myös nämä työkalut, koska ne ovat oleellisessa osassa kokonaiskuvaa katsottaessa.

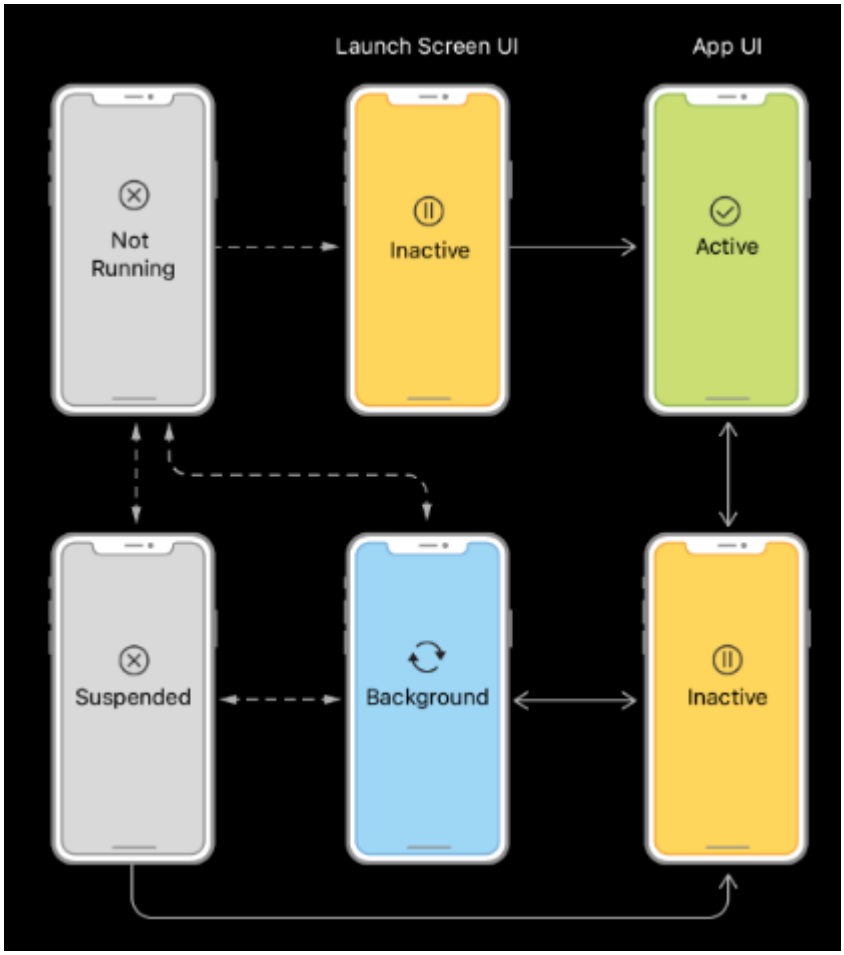
Xcode on Applen oma kehitysympäristö, jolla voi kehittää sovelluksia kaikille Applen alustoille. Lisäksi se on ainut virallinen työkalu sovellusten rakentamiseksi ja julkaisemiseksi Applen App Storessa. Xcode on asennettavissa Applen MacOS laitteille ilmaiseksi ja se tukee monia suosittuja ohjelmointikieliä, kuten Swift, Objective-C, Objective-C++, C, C++, Java, Python ym. (Ekren 2023.)

Swift on Applen kehittämä avoimen lähdekoodin ohjelmointikieli, joka esiteltiin vuonna 2014 Applen maailmanlaajuisessa kehittäjien konferenssissa (*Worldwide Developers Conference, WWDC).* Swift on kehitetty suorituskyky edellä ja Applen mukaan Swift onkin 2,6 kertaa nopeampi kuin Objective-C ja 8,4 kertaa nopeampi kuin Python. Swift on ohjelmointikielenä helppo oppia sen puhtauden ja yksinkertaisen syntaksin ansiosta. Swift on myös yhteensopiva aiemmin iOS-kehitykseen käytetyn Objective-C:n kanssa ja niitä voi käyttää keskenään samassa projektissa. (AltexSoft 2021.)

Työkaluna Xcode on välttämätön osa mitä tahansa Apple-kehitystä, koska ilman sitä ei pysty rakentamaan sovelluksia niin, että ne voitaisi jakaa käyttäjille. Xcoden kanssa käytettäväksi ohjelmointikieleksi valikoitui kuitenkin Swift sen aloittelijaystävällisen ja yksinkertaisen syntaksin vuoksi, jotta uuden kielen opetteluun ei menisi liikaa aikaa.

3.3 iOS-kehitys

Applen dokumentaation mukaan sovelluksen aktiivisuustila (kuva 4) määrittää, mitä se voi milläkin hetkellä tehdä ja mitä ei. Etualalla olevalla sovelluksella on aina etusija järjestelmäresursseihin ja samalla taustalla pyörivän sovelluksen on tehtävä mahdollisimman vähän työtä, mieluiten ei mitään. (Managing… s.a.)



Kuva 4. iOS sovelluksen aktiivisuustilat (Managing… s.a.)

Chinthaka Perera (2019) kuvailee blogikirjoituksessa näitä kuvassa 4 esiteltyjä tiloja hieman laajemmin. Ei käynnissä (engl. *not running*) on tila, jossa sovellus ei nimensä mukaisesti ole käynnissä, eikä suorita koodia taustalla. Passiivinen (engl. *inactive*) on siirtymätila, jossa sovellus käytetään sen siirtyessä ei käynnissä olevasta aktiiviseksi, aktiivisesta taustalle tai taustalta aktiiviseksi. Passiivisessa tilassa esimerkiksi sovelluksen käyttöliittymä ei ole käytettävissä, mutta sovellus pystyy suorittamaan koodia. Aktiivinen (engl. *active*) sen sijaan on sovelluksen päätila, jossa sovellus on käyttäjän näytöllä auki, käyttöliittymä on käytettävissä ja sovellus suorittaa koodia. Taustalla (engl. *background*) on tila, johon siirrytään poistuttaessa sovelluksesta. Tässä tilassa sovellus pystyy suorittamaan koodia, mutta vain hetken ennen kuin se siirretään joko ei käynnissä olevaksi tai keskeytetyksi (engl. *suspended*). Kun sovellusta ei poisteta taustalla olevien sovellusten listasta, se siirtyy keskeytetyksi, joka on tila, jossa sovellus on edelleen taustalla, mutta se ei suorita enää koodia, eikä näin ollen myöskään vie resursseja.

iOS-sovelluksen normaaliin sykliin kuuluu siis se, että se siirtyy lopulta keskeytetyksi, jolloin käyttöjärjestelmä pystyy hallinnoimaan resursseja sen tarpeiden mukaan. Apple tarjoaa kuitenkin muutamia taustasuoritustiloja, joiden käyttöönottamisella sovellus saa mahdollisuuden suorittaa koodia taustalla. Näitä taustasuoritustiloja ovat esimerkiksi äänentoisto, sijaintipäivitysten vastaanottaminen, ajastetut taustatehtävät ja ulkoisen laitteen kanssa viestiminen esimerkiksi Bluetoothin kautta. Taustasuoritustiloja tulisi kuitenkin käyttää säästeliäästi, koska niiden käyttö voi vaikuttaa negatiivisesti puhelimen suorituskykyyn ja akun käyttöikään. (Configuring… s.a.)

Taustatehtäville on kahdenlaisia taustasuoritustiloja: pienille, lyhytkestoisille tehtäville tarkoitettu *background fetch* ja mahdollisesti pitkään vieville tehtäville, kuten suurten tiedostojen lataamiselle tarkoitettu *background processing*. Background Tasks -sovelluskehyksestä löytyy molemmille taustasuoritustiloille niitä vastaavat taustatehtävät BGAppRefreshTask ja BGProcessingTask. Kummankin tyyppisen taustatehtävän voi ajastaa. (Using background… s.a.) Applen WWDC tallenne vuodelta 2020 (Apple Developer Videos 2020) kuitenkin muistuttaa, että järjestelmä ajastaa tehtävät parhaaksi näkemällään tavalla huolimatta siitä, kuinka usein sovelluksen kehittäjä on pyytänyt tehtäviä suoritettavaksi.

Sijaintipäivitysten vastaanottamiselle on siis oma taustasuoritustilansa. Tämän tilan aktivoimalla järjestelmä saa tiedon olla siirtämättä sovellusta keskeytetyksi ja sovellus voi vastaanottaa sijaintipäivityksiä reaaliajassa. Ilman tätä taustasuoritustilaa sijaintipäivitykset asetetaan jonoon ja toimitetaan vasta, kun sovellus seuraavan kerran suorittaa koodia joko aktiivisessa tilassa tai taustalla. (Handling location… s.a.) Sijaintitietojen vastaanottamisen implementoimiseksi täytyy käyttää Core Location -sovelluskehystä, joka tarjoaa palveluja, joilla voidaan määritellä mm. laitteen sijaintia, korkeutta ja suuntaa. Core Location kerää dataa käyttäen kaikkia laitteen saatavilla olevia komponentteja kuten mm. Wi-Fi-yhteyttä, GPS:ää, Bluetoothia, magnetometriä ja barometriä. Sovelluskehyksen kautta on mahdollista saada suuria tai pieniä muutoksia käyttäjän sijainnissa ja päivitysten tarkkuutta voidaan säätää tilanteen mukaan. (Core Location s.a.)

Applen Core Motion -sovelluskehys puolestaan raportoi erilaista liikkeeseen ja ympäristöön liittyvää dataa riippuen siitä, millainen laitteisto puhelimessa on sisällä. Dataa voi saada joko raakadatana tai prosessoituna, mutta monet sovelluskehyksen palvelut tarjoavat molempia. Core Motion sisältää keinoja erilaisten puhelimen sisäisten anturien datan keräämiseen. Näitä antureita ovat mm. kiihtyvyysanturi, gyroskooppi, askelmittari, magnetometri ja barometri. Lisäksi on mahdollista saada historiatietoja esimerkiksi liikkeentunnistuksesta vastaavalta CMMotionActivityManager-luokalta. (Core Motion s.a.) CMMotionActivityManager-luokan metodien avulla voidaan havaita, käveleekö, juokseeko, onko käyttäjä liikkumatta tai liikkuuko käyttäjä jollakin kulkuneuvolla. (CMMotionActivityManager s.a.)

Core Motion -sovelluskehyksestä löytyvällä CMMotionManager-luokalla voi joko kerätä kiihtyvyysanturista dataa reaaliajassa tai tallentaa kerättyä dataa myöhemmin käytettäväksi. (CMMotionManager s.a.) Core Motion -sovelluskehys tarjoaa myös toisen tavan nauhoittaa kiihtyvyysanturia: instanssimetodi recordAccelerometer(forDuration), jolla voi nauhoittaa 50Hz taajuudella maksimissaan 12 tuntia. Nauhoituksen pituuden voi itse määritellä 12 tuntiin asti, mutta nauhoitustaajuutta ei voi säätää, vaan se on ennalta määritelty. Metodi kuitenkin vähän rikkoo aikaisemmin opittuja säännönmukaisuuksia sovelluksen toimimisesta taustalla, sillä se kykenee nauhoittamaan kiihtyvyysanturia myös keskeytetty- ja ei käynnissä -tiloissa. (recordAccelerometer… s.a.)

4 ENSIMMÄINEN SOVELLUSVERSIO

Tässä luvussa käsitellään ensimmäisen sovellusversion kehityksen vaiheita sekä sen testausta. Ensimmäisen sovellusversion tarkoituksena oli siis pyrkiä nauhoittamaan iPhonen kiihtyvyysanturia vuorokauden ympäri, jotta nähdään, kykeneekö puhelin nauhoittamaan kiihtyvyysanturia katkeamatta vai sulkeeko käyttöjärjestelmä sovelluksen, jos se vie liikaa resursseja, tai onko datassa jotain puutteita. Sovelluksen testaukseen osallistui 11 iPhone-käyttäjää ja lisäksi toimeksiantajalta saatiin testausta varten yksi iPhone, jotta nähdään, onko mitään eroa sovelluksen toimivuudessa silloin, kun se on päivittäisessä käytössä, verrattuna puhelimeen, joka on vain testikäytössä. Testaukseen saatiin laaja skaala puhelimia eri sukupolvista: 2 x iPhone 6s, iPhone 14 Pro Max, 2 x

iPhone 12, iPhone 7, iPhone 14 Pro, iPhone XR, iPhone 15 Pro, iPhone 7 Plus, iPhone 12 Mini sekä 3. sukupolven iPhone SE.

4.1 Kehitys

Ensimmäisen sovellusversion kehittäminen aloitettiin tutustumalla Applen dokumentaatioon eri sovelluskehyksistä ja niiden käytöstä. Dokumentaation avulla rakennettiin pieniä prototyyppejä, joilla testattiin, miten toiminnallisuuksien käytännön toimivuutta. Näiden prototyyppien ja niiden testauksen pohjalta rakennettiin toiminnallisuus varsinaiseen sovellusversioon, joka jaettiin testiryhmälle.

Ensimmäisellä prototyypillä testattiin kiihtyvyysanturin nauhoitusta Core Motion -sovelluskehyksen recordAccelerometer-metodilla ja nauhoituksen uudelleen aloittamista ajastetuilla taustatehtävillä, koska kyseisellä metodilla voitiin nauhoittaa maksimissaan vain 12 tuntia kerrallaan. Prototyyppiä varten rakennettiin Flutter-rajapinnalla simppeli käyttöliittymä, jossa ei oikeastaan ollut muuta kuin painike kerätyn datan lähettämiseksi palvelimelle. Muu toiminnallisuus toteutettiin Swift-ohjelmointikielellä sovelluksen natiivipuolelle. Prototyyppi rakennettiin toimimaan nauhoituksen osalta itsenäisesti: sovellus aloitti nauhoituksen itse, kun käyttäjä poistui sovelluksesta ja sovellus siirtyi taustalle (kuva 5), ja nauhoitus lopetettiin, kun nauhoitukselle määritelty maksimiaika tuli umpeen, käyttäjä avasi sovelluksen uudestaan näytölle, jolloin se siirtyi taas aktiiviseksi, tai kun sovellus siirtyi ei käynnissä -tilaan joko käyttäjän sulkiessa sovelluksen kokonaan tai käyttöjärjestelmän sulkiessa sovelluksen resurssien vapauttamiseksi (kuva 6).

```
override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
) -> Bool {
    GeneratedPluginRegistrant.register(with: self)
    BackgroundRecorderPlugin.register(with: registrar(forPlugin: "recorder")!)
    BGTaskScheduler.shared.register(forTaskWithIdentifier: "com.example.iosBgTest.BgTask", using: nil) {task in
        self.handleBackgroundTask(task: task as! BGAppRefreshTask)
    }
    if AVAudioSession.sharedInstance().isOtherAudioPlaying{…
    } else {…
    }
    self.audioPlayer = AudioPlayer()
    return super.application(application, didFinishLaunchingWithOptions: launchOptions)
}

override func applicationDidEnterBackground(_ application: UIApplication){
    scheduleBackgroundTask()
}

override func applicationWillEnterForeground(_ application: UIApplication) {…
}

override func applicationWillTerminate(_ application: UIApplication) {…
}

func scheduleBackgroundTask(){
    print("scheduling the background task")
    let request = BGAppRefreshTaskRequest(identifier: "com.example.iosBgTest.BgTask")
    request.earliestBeginDate = Date(timeIntervalSinceNow: 60 * 60)

    do {
        recordStartingTime = Int64(Date().timeIntervalSince1970 * 1000)
        print("Record starting time: \(recordStartingTime)")
        if CMSensorRecorder.isAccelerometerRecordingAvailable(){
            DispatchQueue.global(qos: .background).async{
                self.sensorRecorder.recordAccelerometer(forDuration: 60 * 60 * 12)
                self.audioPlayer?.playSound(soundName: "beepbeepbeep")
            }
        }
        try BGTaskScheduler.shared.submit(request)
        print("task scheduled, recording started")
    } catch {
        print("Unable to schedule background task: \(error)")
    }
}

func handleBackgroundTask(task: BGAppRefreshTask) {
    recordEndingTime = Int64(Date().timeIntervalSince1970 * 1000)
    print("Record ending time: \(recordEndingTime)")
    saveAccelerometerData(startTimeMillis: recordStartingTime, endTimeMillis: recordEndingTime)

    print("Starting new recording")
    self.scheduleBackgroundTask()
    task.setTaskCompleted(success: true)
}
```

Kuva 5. Ensimmäisen prototyypin taustatehtävä nauhoituksen aloittamiselle

Kuvasta 5 voidaan nähdä mekaniikkaa kiihtyvyysanturin nauhoituksen aloittamiselle ja taustatehtävän ajastamiselle. Sovelluksen käynnistyessä rekisteröidään ajastettava taustatehtävä uniikilla tunnisteella ja määritellään metodi, jota kutsutaan, kun taustatehtävä suoritetaan. Tämä taustatehtävä on tyyppiä BGAppRefreshTask ja sen toimimiseksi täytyy ottaa myös käyttöön background fetch -taustasuoritustila. Kun sovellus siirtyy taustalle, se suorittaa automaattisesti applicationDidEnterBackground-metodin, jonka sisällä kutsutaan scheduleBackgroundTask-metodia. scheduleBackgroundTask-metodi

määrittelee kuinka nopeasti seuraava taustatehtävä tulisi suorittaa, jonka jälkeen se asettaa nauhoitukselle aloitusajan ja aloittaa 12 tunnin nauhoituksen. Tämän jälkeen metodi lähettää käyttöjärjestelmälle pyynnön suorittaa taustatehtävä aikaisintaan aiemmin määritellyn ajan kuluttua. Täytyy muistaa, että tämä on nimenomaan pyyntö toteuttaa taustatehtävä tietyn ajan kuluttua, koska käyttöjärjestelmä säätelee itse resurssien käyttöä ja suorittaa tehtäviä parhaaksi kokemallaan hetkellä. Kun taustatehtävä sitten joskus järjestelmän niin salliessa suoritetaan, kutsutaan aiemmin tehtävälle määriteltyä handleBackgroundTask-metodia, joka asettaa nauhoitukselle lopetusajan ja tallentaa aloitus- ja lopetusaikojen välisen nauhoitteen. Tämän jälkeen metodi kutsuu scheduleBackgroundTask-metodia, jossa aloitetaan uusi nauhoite ja ajastetaan taustatehtävä uudelleen.

```
override func applicationWillEnterForeground(_ application: UIApplication) {
    recordEndingTime = Int64(Date().timeIntervalSince1970 * 1000)
    print("Record ending time: \(recordEndingTime)")
    saveAccelerometerData(startTimeMillis: recordStartingTime, endTimeMillis: recordEndingTime)
    self.audioPlayer?.playStopRecordingSound()
}

override func applicationWillTerminate(_ application: UIApplication) {
    recordEndingTime = Int64(Date().timeIntervalSince1970 * 1000)
    print("Record ending time: \(recordEndingTime)")
    saveAccelerometerData(startTimeMillis: recordStartingTime, endTimeMillis: recordEndingTime)
    self.audioPlayer?.playStopRecordingSound()
}
```

Kuva 6. Ensimmäisen prototyypin mekaniikka nauhoituksen lopettamiselle, kun sovellus siirtyy aktiiviseksi, tai kun sovellus suljetaan

applicationDidEnterBackground-metodin lisäksi sovellus suorittaa automaattisesti kuvassa 6 näkyvät metodit applicationWillEnterForeground, kun sovellus on siirtymässä aktiiviseksi ja applicationWillTerminate, kun sovellus suljetaan. Molemmissa tapauksissa nauhoitukselle asetetaan loppumisaika ja tallennetaan nauhoitus samoin kuin taustatehtävän aikana, mutta tässä vaiheessa ei kutsuta uudestaan scheduleBackgroundTask-metodia, vaan se toistetaan vasta sitten, kun sovellus siirtyy uudestaan taustalle. Jos nauhoitus lopetetaan siksi, että sovellus suljetaan joko käyttäjän tai käyttöjärjestelmän toimesta, on nauhoituksen uudelleenkäynnistämiseksi avattava sovellus uudestaan, jotta se voidaan taas saattaa taustatilaan sovelluksesta poistuttaessa.

Vaikka kiihtyvyysanturin nauhoittaminen tällä tyylillä säästäisi resursseja nauhoituksen toimiessa myös ei aktiivisessa tilassa ja olisi näin ollen paras vaihtoehto, osoittautui se hyvin nopeasti kuitenkin käytännössä toimimattomaksi. Taustatehtävä, jolle lopulta pyydettiin suoritusaikaa optimistisesti tunnin välein (kuva 5, s. 18), jotta tehtävä tapahtuisi mieluummin liian usein kuin liian harvoin, ei sitten saanutkaan järjestelmältä lupaa niin usein kuin toivottiin. Alkuvaiheessa taustatehtävän suoritusta testattiin jopa minuutin ajastuksella ja virheenkorjaustilassa sovellusta ajettaessa taustatehtävä tapahtui tällöin yleisesti jossakin kymmenen minuutin ja kahden tunnin välillä, ja tämä vaikutti suhteellisen lupaavalta. Ajastus nostettiin tämän jälkeen tuntiin ja virheenkorjaustilassa kaikki näytti edelleen hyvältä, mutta kun sovelluksesta rakennettiin valmis versio ja sitä testattiin, tapahtui taustatehtävä enää kerran päivässä, yleensä myöhään illalla. Kahden taustatehtävän suorituksen väliin jäi siis noin 24 tuntia, jolloin puolet päivästä jäi nauhoittamatta, ja sekin puoli, joka nauhoitettiin, oli yöllä, kun puhelin ei ollut käytössä. recordAccelerometer-metodin käytöstä lopulliseen versioon päätettiin siis luopua heti ensimmäisten kokeilujen jälkeen.

Seuraava prototyyppi rakennettiin, jotta voitiin testata, miten sijaintitietojen kerääminen käytännössä toimii, koska Core Motion -sovelluskehyksen CMMotionManager-luokan tarjoama kiihtyvyysanturin nauhoittaminen onnistuu vain, kun sovellus on joko aktiivisena tai taustalla ja tarvittiin jokin keino pitää sovellus auki myös taustalla. Koska Core Motion -sovelluskehyksen tarjoama merkittävien sijaintimuutosten (engl. *significant location changes*) tarkkailu tarjoaa hetken suoritusaikaa taustalla aina, kun sijainti muuttuu vähintään 500 m ja siirtää sen jälkeen sovelluksen takaisin keskeytetyksi, kunnes uusi sijaintipäivitys saadaan, ei sitä voitu tässä tapauksessa käyttää sovelluksen pitämiseen auki taustalla, joten koko projektissa päädyttiin käyttämään jatkuvia sijaintipäivityksiä. Tässä prototyypissä käytettiin kuitenkin vielä recordAccelerometer-metodia, koska siitä oli olemassa jo teoriassa toimiva toteutus kiihtyvyysanturin nauhoittamiseen ja haluttiin keskittyä testaamaan muita toiminnallisuuksia. Myös tähän prototyyppiin rakennettiin Flutter-rajapinnalla yksinkertainen käyttöliittymä ja siinä oli ensimmäisen prototyypin tapaan ainoastaan painike kerätyn datan lähettämiselle. Tässä vaiheessa oli tarkoitus testata, miten sijaintipäivitysten käyttö saataisiin optimoitua niin, että sovellus kuluttaa mahdollisimman vähän akkua kokeilemalla, miten akunkulutus muuttuu, jos Core Location

-sovelluskehyksen tarjoaman sijaintimanagerin muuttujien arvoja (kuva 7) muutetaan. Koska haluttiin käyttää sijaintitietoja Core Location -sovelluskehyksen kautta, täytyi myös muistaa aktivoida sijaintitietojen käytön salliva taustasuoritustila, jotta voitiin kerätä sijaintitietoja myös taustalla.

```
func initializeLocationManager() {
    locationManager.delegate = self
    locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters
    locationManager.distanceFilter = CLLocationDistance(10)
    locationManager.requestAlwaysAuthorization()
    locationManager.allowsBackgroundLocationUpdates = true
    locationManager.pausesLocationUpdatesAutomatically = false;
    locationManager.startUpdatingLocation()
}
```

Kuva 7. Toisen prototyypin sijaintimanagerin muuttujien alustaminen

Ennen sijaintipäivitysten vastaanottamista voi sijaintimanagerille alustaa erilaisia, esimerkiksi kuvassa 7 näkyviä arvoja. Jotta sijaintia voidaan käyttää myös silloin, kun sovellus on taustalla, pelkkä taustasuoritustilan aktivoiminen ei riitä, vaan sijaintimanagerille täytyy myös kertoa, että se saa käyttää sijaintia taustalla. Käyttäjältä pyydetään lupa sijaintitietojen käyttämiseen requestAlwaysAuthorization-metodilla ja allowsBackgroundLocationUpdates-muuttuja asetetaan todeksi. Samalla pausesLocationUpdatesAutomatically-muuttuja asetetaan epätodeksi, koska muuten sijaintipäivitykset katkeavat automaattisesti aina kun mahdollista akun säästämiseksi, eikä sijaintipäivityksiä voi jatkaa ilman käyttäjän panosta. (pausesLocationUpdatesAutomatically s.a.) Lisäksi sijaintimanagerille voi asettaa esimerkiksi toivotun tarkkuuden sijaintipäivityksille, sekä etäisyyssuodattimen, jolla voi määritellä kuinka paljon käyttäjän täytyy liikkua, jotta seuraava sijaintipäivitys saadaan. (distanceFilter s.a.) Prototyyppiä testattiin useilla eri tarkkuus- ja etäisyyssuodatinarvoilla, mutta paras akunsäästö syntyi, kun tarkkuus asetettiin huonoimmaksi ja etäisyyssuodatin suurimmaksi mahdolliseksi. Näin sovellus haki sijaintia mahdollisimman harvoin, mutta pysyi silti auki taustalla.

Sijainnin lisäksi prototyypillä testattiin, onko mahdollista kutsua natiivipuolelta Flutter-rajapinnan funktioita virallinen ensimmäinen sovellusversio mielessä, koska sovelluksen datan lähetys oli rakennettu Flutter-rajapinnalla ja haluttiin ajastaa taustatehtävä natiivipuolelta kutsumaan Flutter-puolta, josta lähetys

tehtäisi. Normaalisti keskustelu Flutter-puolen ja natiivipuolen välillä tehdään käyttäen metodikanavaa, joka kutsuu natiivipuolen metodeja, kun jokin metodi suoritetaan Flutter-puolella ja nyt haluttiin tietää voiko metodikanavia käyttää myös toiseen suuntaan. Natiivipuolelle rekisteröitiin Swift-ohjelmointikielellä samanlainen ajastettu taustatehtävä, kuin ensimmäisessä prototyypissä, mutta tällä kertaa käyttäen tehtävätyyppiä BGProcessingTask (kuva 8). BGProcessingTask alustettiin täysin samalla tavalla kuin BGAppRefreshTask, kuvassa 5 (s. 18) näkyvän taustatehtävän metodin käyttämä tehtävätyyppi vain vaihdettiin BGProcessingTask-tyyppiseksi. Taustasuoritustilana täytyi tässä vaiheessa käyttää background fetch -taustasuoritustilan sijasta background processing -taustasuoritustilaa.

```swift
override func applicationDidEnterBackground(_ application: UIApplication){
    startPeriodicTask()
}

override func applicationWillEnterForeground(_ application: UIApplication) {
}

override func applicationWillTerminate(_ application: UIApplication) {
}

func handleBackgroundTask(task: BGProcessingTask){
    print("starting background task")
    DispatchQueue.main.async{
        iOSPlugin.channel?.invokeMethod("backgroundTask", arguments: nil)
    }

    self.startPeriodicTask()
    task.setTaskCompleted(success: true)
}

func startPeriodicTask(){
    print("Scheduling a background task")
    let request = BGProcessingTaskRequest(identifier: "com.example.iosLocationTest.iosLocTest")
    request.earliestBeginDate = Date(timeIntervalSinceNow: 60)
    request.requiresNetworkConnectivity = false
    request.requiresExternalPower = false

    do{
        try BGTaskScheduler.shared.submit(request)
    } catch {
        print("Couldn't schedule background task: \(error)")
    }
}
```

Kuva 8. Toisen prototyypin taustatehtävä

Kuvassa 8 näkyvälle taustatehtävälle ajastettiin alkuun tehtävien väliksi yksi minuutti, jotta toimivuuden testaamiseksi parhaimmassa tapauksessa ei tarvinnut odottaa kuin minuutti, ennen kuin järjestelmä kutsui taustatehtävää. Kun handleBackgroundTask-metodia kutsutaan, se lähettää tiedon erillisessä lisäosassa olevalle metodikanavalle (kuva 9), että taustatehtävää on kutsuttu.

Flutter kuuntelee metodikanavaa ja kun se saa tiedon, mitä metodia natiivi-puolella on kutsuttu (kuva 10), voidaan Flutter-puolella suorittaa eri asioita kutsutun metodin perusteella.

```
public class iOSPlugin: NSObject, FlutterPlugin {
    static var channel: FlutterMethodChannel?
    public static func register(with registrar: FlutterPluginRegistrar){
        channel = FlutterMethodChannel(
            name: "plugin",
            binaryMessenger: registrar.messenger()
        )
        let instance = iOSPlugin()
        registrar.addMethodCallDelegate(instance, channel: channel!)
    }
```

Kuva 9. Toisen prototyypin metodikanava

Kuvassa 9 näkyy sovellukseen kirjoitettu lisäosa. Ihan kuten taustatehtävät, myös metodikanavat täytyy rekisteröidä ja tämä tapahtuu lisäosassa. Metodikanava rekisteröidään uniikilla nimellä ja sille annetaan viestinvälittäjä.

```
@override
void initState() {
  super.initState();
  WidgetsBinding.instance.addObserver(this);
  Permission.sensors.request().then((state) {
    if (state.isGranted) {
      print('Permissions granted');
    } else {
      print('Permissions denied');
    }
  });
  setUpMethodChannelListener();
}

void setUpMethodChannelListener() {
  channel.setMethodCallHandler((call) async {
    if (call.method == "backgroundTask") {
      print("Seems to be working");
    }
  });
}
```

Kuva 10. Flutter-puolella oleva metodikanavan kuuntelija

Flutter-puolelle tarvitaan siis kuuntelija, joka kuuntelee metodikanavan viestejä. Kuvassa 10 esitellään setUpMethodChannelListener-metodi, joka kuuntelee natiivipuolen taustatehtävää. Kuuntelijaa kutsutaan heti sovelluksen alustavassa initState-metodissa, kun sovellus käynnistyy, jolloin se kuuntelee metodikanavaa heti sovelluksen käynnistymisestä siihen asti, kunnes sovellus suljetaan. Tässä prototyypissä ei kutsuttu varsinaisesti mitään erityistä metodia Flutter-puolella, vaan yksinkertaisesti tulostettiin virheenkorjauslokiin tekstiä, jotta tiedettiin Flutter-rajapinnan saaneen tiedon taustatehtävästä. Testatessa sovellus tulosti lokitekstin aina, kun taustatehtävä suoritettiin, eli selkeästi viestintä myös natiivipuolelta Flutter-rajapinnalle toimii, vaikkakin taustatehtävien tapahtuminen itsessään on epävarmaa ja kaikki riippuu käyttöjärjestelmästä.

Varsinainen ensimmäinen sovellusversio isompaa testausta varten rakennettiin siis aiemmin esiteltyjen prototyyppien pohjalta käyttäen hyvin pitkälti samanlaisia metodeja. Mekaniikat luotiin täysin erilliseen lisäosaan, jotta se on tarvittaessa liitettävissä sellaisenaan myös muihin projekteihin. Sijaintimanageri alustettiin samoilla muuttujilla, kuin toinen prototyyppi (kuva 7, s. 20) ja tarkkuus asetettiin mahdollisimman huonoksi, sekä etäisyyssuodatin mahdollisimman suureksi akunkulutuksen minimoimiseksi. Lisäksi sovellukseen rekisteröitiin tunnin välein tapahtuvaksi ajastettu taustatehtävä, jonka tarkoituksena oli laukaista metodi Flutter-puolella tallennetun datan lähettämiseksi palvelimelle, käyttäen metodikanavaa. Metodikanava implementoitiin myös samoin, kuin toisessa prototyypissä (kuva 9, s. 22). Myös Flutter-puolen metodikanavan kuuntelijan implementoinnissa noudatettiin samaa mallia, kuin toisessa prototyypissä. Tällä kertaa tosin kutsuttiin Flutter-puolen metodia, joka lähettää kerätyn datan palvelimelle sen sijaan, että tulostettaisi lokiin tekstiä. Koska tarkoituksena on kehittää monialustainen sovellus ja sen Android-puoli oli jo kehityksessä, oli sovelluksen Flutter-puoli jo rakennettu. Käyttöliittymässä oli painike nauhoituksen aloittamiselle ja lopettamiselle, joten iOS-natiivipuolelle implementoitiin myös metodit nauhoituksen aloittamiselle ja lopettamiselle. Kun nauhoitus aloitetaan, alustetaan sijaintimanageri ja aloitetaan sijaintietojen päivitys, sekä kiihtyvyysanturin nauhoitus. Kun nauhoitus puolestaan lopetetaan, sijaintitietojen päivitys ja anturin nauhoitus lopetetaan. Flutter-puo-

lelle täytyi myös implementoida omat metodit nauhoituksen aloittamiselle ja lopettamiselle iOS-puolelle, koska se toimii hieman eri tavalla kuin Android-puoli (kuva 11).

```
Future<bool> startiOS() async {
  bool result = true;
  final locationPermissionRequestResult =
      await Permission.locationAlways.request();
  final activityRecognitionResult = await Permission.sensors.request();
  if (!locationPermissionRequestResult.isGranted ||
      !activityRecognitionResult.isGranted) {
    result = false;
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text(
              'Application needs these permissions in order to work properly.'),
        ), // SnackBar
      );
    }
  }

  logger.info('Starting iOS background recording.');
  XeloBackground.instance.startBackgroundRecording();
  return result;
}

Future<void> stopiOS() async {
  XeloBackground.instance.stopBackgroundRecording();
}
```

Kuva 11. Metodit nauhoituksen aloitukselle ja lopetukselle iOS-alustalle Flutter-puolella

Kuvassa 11 on esillä Flutter-puolen metodit nauhoituksen aloitukselle ja lopetukselle. Sovelluksen täytyy kysyä lupa käyttäjältä anturin ja sijaintitietojen käyttämiseen ennen kuin niitä voidaan käyttää ja vasta sen jälkeen voidaan nauhoitus aloittaa. Android-puolella tarvitaan myös lupa akunkäytön optimoinnille, mutta tällaista vaihtoehtoa ei ole iOS-luvissa, joten tämä metodi luotiin käytettäväksi ainoastaan silloin, kun käyttäjällä on iPhone. XeloBackground on se aiemmin mainittu erillinen lisäosa, johon kaikki natiivipuolen toiminnallisuudet toteutettiin. Nauhoituksen aloitus- ja lopetusmetodeissa täytyy kutsua lisäosan käyttämiä metodeja, jotka sitten toteuttavat halutut mekaniikat suoraan natiivipuolelta. Koska ensimmäisen prototyypin testauksessa todettiin, että recordAccelerometer-metodi ei ollut soveltuva kiihtyvyysanturin nauhoitukseen tässä projektissa, käytettiin varsinaisissa isommin testattavissa sovellusversioissa Core Motion -sovelluskehyksestä löytyvää CMMotionManager-luokkaa

kiihtyvyysanturin nauhoittamiseen (kuva 12). Tällä pystyi lisäksi myös tallentamaan datan reaaliajassa, kun taas recordAccelerometer-metodia käyttämällä se täytyi hakea jälkikäteen, vaikka itse nauhoitus tapahtuikin reaaliajassa.

```
func startRecording(){
    print("Starting accelerometer updates")
    motionManager.accelerometerUpdateInterval = 1.0 / Double(samplingPeriodHz)
    if motionManager.isAccelerometerAvailable {
        motionManager.startAccelerometerUpdates(to: .main) {(accelerometerData, error) in
        let acceleration = accelerometerData!.acceleration
        self.saveAcceleration(accelerometerData: acceleration)
        }
    }
}

public func saveAcceleration(accelerometerData: CMAcceleration) {
    let systemCurrentTimeMillis = Int64(Date().timeIntervalSince1970 * 1000)

    let x = Float(accelerometerData.x)
    let y = Float(accelerometerData.y)
    let z = Float(accelerometerData.z)
    let m = sqrt(x * x + y * y + z * z)

    var timestamp = systemCurrentTimeMillis

    print(timestamp, x, y, z, m)

    assignToArray(array: &spikeData, x: x, y: y, z: z, m: m)
    let spikeDataClone = spikeData
    let insertSaveResult = db.saveRecord(timestamp: timestamp, arr: spikeDataClone)
    print(insertSaveResult)
}
```

Kuva 12. Kiihtyvyysanturin nauhoitus natiivipuolella käyttäen CMMotionManager-luokkaa

Kuvassa 12 näkyvät startRecording- ja saveAcceleration-metodit vastaavat kiihtyvyysanturin nauhoittamisesta ja datan tallentamisesta. Kun nauhoitus aloitetaan, sille voi ensin määritellä halutun päivitysvälin ja tässä sille asetetaan päivitysväliksi ennalta samplingPeriodHz-muuttujaan asetettu taajuus hertseissä. Sen jälkeen tarkistetaan, onko laitteessa kiihtyvyysanturi, jotta sitä voidaan käyttää. startAccelerometerUpdates-metodi aloittaa kiihtyvyysanturin datan päivittämisen annetulla päivitysvälillä ja toisin kuin recordAccelerometer-metodi, se jatkaa niin kauan, kunnes päivitykset lopetetaan manuaalisesti. Jokainen datan pala, joka sisältää x-, y- ja z-akselin kiihtyvyydet tallennetaan listaan, johon tallennetaan lisäksi akseleiden välisistä kiihtyvyyksistä laskettu magnitudi. Lisäksi tallennetaan aikaleima hetkestä, jolta data on saatu, jonka jälkeen lista kiihtyvyyksistä ja magnitudista tallennetaan aikaleiman kanssa puhelimelle luotuun tietokantaan, josta ne myöhemmin lähetetään Flutter-puolelta palvelimelle.

4.2 Testaus

Ensimmäinen testausviikko toteutettiin 100Hz nauhoitustaajuudella. Sovellus kerkesi nauhoittamaan testaajien puhelimilla viikonlopun yli, kunnes tarkistettiin, onko puhelimet lähettäneet palvelimelle mitään. Yllätykseksi palvelin oli tyhjä, joten toimeksiantajan tarjoamalla testipuhelimella yritettiin lähettää data manuaalisesti, mutta manuaalinenkaan lähetys ei toiminut. Todennäköisesti lähetettävän datan määrä oli liian suuri tehtäväksi samaan aikaan, kun puhelimelle tallennetaan jatkuvasti uutta dataa ja lähetys meni jumiin. Tästä syystä datan lähetystä Flutter-puolella muutettiin niin, että nauhoitus katkaistiin datan lähetyksen ajaksi ja kun lähetys oli valmis, jatkettiin nauhoitusta. Aikaisemmissa sisäisissä testeissä huomattiin, että uuden sovellusversion asentaminen poisti aiemmin nauhoitetun datan, jos niitä ei ollut keretty vielä lähettää palvelimelle, mutta kun palvelimelle viimein saapui testikäyttäjiltä dataa, oli siellä positiiviseksi yllätykseksi dataa myös ajalta ennen datan lähetyksen korjausta ja uuden sovellusversion asentamista. Kaikilla puhelimilla data ei ollut kuitenkaan säilynyt, joten siinä miten data säilyy sovellusta päivittäessä voi olla eroja eri mallien välillä tai siinä, onko käyttäjällä käytössä iCloud varmuuskopiointia varten.

Ensimmäisen viikon aikana kaikissa puhelimissa käyttöjärjestelmä oli yleisesti sulkenut sovelluksen kerran päivässä ja loppuviikosta kumpikin iPhone 12, iPhone 6s ja iPhone 12 Mini sulkeutuivat kahdesti päivän aikana. Suurin osa puhelimista kykeni kuitenkin nauhoittamaan kiihtyvyysanturia koko sen ajan, minkä käyttöjärjestelmä oli sovellukselle suonut. Muutaman vanhemman puhelinmallin lähettämässä datassa oli yhdestä kahteen pientä, maksimissaan muutaman sekunnin katkoa nauhoituksessa (kuva 13), mutta toinen iPhone 6s lähetti dataa, jossa oli yhden päivän aikana jopa 14 pientä katkoa nauhoituksessa. Vain iPhone 14 Pro, iPhone 14 Pro Max, iPhone 15 Pro, iPhone 12 Mini ja iPhone 7 lähettivät dataa automaattisesti ensimmäisen viikon aikana, muita testaajia täytyi pyytää lähettämään data manuaalisesti. Yksikään automaattisesti dataa lähettänyt puhelin ei kuitenkaan suorittanut lähetyksestä vastaavaa taustatehtävää joka päivä. Akkua ensimmäinen versio vei todella paljon. Toimeksiantajayrityksen sisäisissä testeissä arvioitiin akunkulutuksen olevan n. 60 % 24 tunnin aikana, kun puhelinta käytettiin vain testaukseen.

Todellisuudessa, kun puhelimia käytettiin muuhunkin, oli akunkulutus tätäkin suurempi testaajien raportoinnin mukaan.

Kuva 13. Pieniä katkoja iPhone 7 Plus puhelimen lähettämässä datassa. Näitä oli myös muiden vanhempien puhelinmallien lähetyksissä.

Toinen testausviikko toteutettiin 50Hz nauhoitustaajuudella. Käyttöjärjestelmän aiheuttamien sovelluksen sulkeutumisten määrä vaihteli toisella viikolla päivittäin eri puhelinmallien välillä. Yleisesti käyttöjärjestelmä sulki sovelluksen kerran tai kaksi päivässä, osassa puhelimista ei välttämättä edes joka päivä, mutta iPhone XR oli yhtenä päivänä sulkenut sovelluksen jopa neljä kertaa. iPhone 7 Plus sulkeutui kerran ensimmäisenä päivänä, mutta ei ollenkaan koko loppuviikossa. Pienten katkojen määrä oli edelleen hyvin pientä suurimmassa osassa puhelimia, mutta iPhone 7 Plus ja kumpikin iPhone 6s pätkivät toisella viikolla huomattavasti enemmän kuin muut, 5–17 kertaa päivän aikana, päivästä riippuen. Kuvassa 13 nähdään dataa, jota iPhone 7 Plus lähetti toisen viikon aikana ja siitä voidaan nähdä, miten noin kahden minuutin aikaikkunassa on ollut jopa 11 pientä katkoa. Toisen viikon aikana dataa lähetti automaattisesti jo useampikin puhelin: iPhone 12 Mini, toinen iPhone 6s, iPhone 15 Pro, iPhone SE, iPhone 14 Pro Max, iPhone XR ja toinen iPhone 12. Lähes jokainen näistä lähetti dataa kaksi tai kolme kertaa viikon aikana, mutta iPhone 12 Mini ja iPhone 15 Pro lähettivät dataa kuutena peräkkäisenä päivänä. Loput datat testaajat lähettivät taas manuaalisesti. Akkua 50Hz versio vei testaajien mukaan jonkun verran vähemmän kuin 100Hz versio, mutta mitään tarkkoja lukuja ei saatu. Kummankaan viikon aikana ei huomattu eroa pelkästään testikäytössä olevan iPhone SE:n ja muiden, normaalissa käytössä olevien puhelinten välillä.

5 TOINEN SOVELLUSVERSIO

Tässä luvussa käsitellään toisen sovellusversion kehityksen vaiheita, sekä sen testausta. Toisen sovellusversion tarkoituksena oli kehittää toiminnallisuus, joka olisi mahdollisimman lähellä tulevan kaupallisen sovelluksen suunniteltua toimintamallia. Sovelluksen tuli optimoida resurssien käyttö ja välttää turhan datan kerääminen nauhoittamalla kiihtyvyysanturia vain silloin, kun käyttäjä kävelee tai juoksee puhelin mukanaan. Sovelluksen testaukseen osallistui sama testiryhmä, kuin ensimmäisenkin version testaukseen.

5.1 Kehitys

Toisen sovellusversion kehittämistä varten edellisessä luvussa mainittuun toiseen prototyyppiin tehtiin hieman muutoksia, joilla testattiin uusia toiminnallisuuksia. Core Motion -sovelluskehyksen sisältämän CMMotionActivityManager-luokan avulla voidaan siis tarkkailla käyttäjän liikettä ja päätellä liikkuuko käyttäjä kävellen, juosten, kulkuneuvolla, vai onko käyttäjä paikallaan. CMMotionActivityManager-luokan testaamista varten prototyyppiin implementoitiin liikkeentunnistus käyttäen luokan sisältämää startActivityUpdates-metodia (kuva 14).

```
func checkForMovement(){
    motionActivityManager.startActivityUpdates(to: .main){ (activity) in
        if let activity = activity{
            if (activity.walking || activity.running){
                self.locationManager.desiredAccuracy = kCLLocationAccuracyBest
                self.locationManager.distanceFilter = kCLDistanceFilterNone
                if !self.isMoving{
                    print("User is moving, starting the recording")
                    print("Confidence: \(activity.confidence)")
                    if CMSensorRecorder.isAccelerometerRecordingAvailable(){
                        DispatchQueue.global(qos: .background).async{
                            self.recordStartingTime = Int64(Date().timeIntervalSince1970 * 1000)
                            print("Record starting time: \(self.recordStartingTime)")
                            self.sensorRecorder.recordAccelerometer(forDuration: 60 * 60 * 12)
                            self.isMoving = true;
                            self.audioPlayer?.playSound(soundName: "beepbeepbeep")
                        }
                    }
                } else {
                    print("User already in movement, continue")
                }
            } else {
                self.locationManager.desiredAccuracy = kCLLocationAccuracyReduced
                self.locationManager.distanceFilter = CLLocationDistanceMax
                if self.isMoving{
                    self.recordEndingTime = Int64(Date().timeIntervalSince1970 * 1000)
                    print("Record ending time: \(self.recordEndingTime)")
                    self.isMoving = false
                    self.saveAccelerometerData(startTimeMillis: self.recordStartingTime, endTimeMillis: self.recordEndingTime)
                } else {
                    print("User is not moving")
                }
            }
        } else {
            print("No motion activity data available")
        }
    }
}
```

Kuva 14. Liikkeentunnistus CMMotionActivityManager-luokan avulla

Kuvassa 14 näkyvä checkForMovement-metodi vastaa liikkeentunnistuksesta ja prototyypissä sitä kutsuttiin heti, kun sovellus käynnistyi. Kun metodi saa tiedon uudesta liikkeestä, se tarkistaa Applen omien algoritmien pohjalta kävi-leekö tai juokseeko käyttäjä, vai vastaako liike jotain muuta aktiviteettia. Jos liike tapahtuu jalan, parannetaan sijainnin tarkkuutta ja poistetaan etäisyys-suodatin käytöstä, jotta sijaintipäivitys saadaan mahdollisimman usein. Jos käyttäjä ei edellisen päivityksen aikaan liikkunut, aloitetaan myös kiihtyvyysan-turin nauhoittaminen ja jos käyttäjä oli liikkeessä myös edellisen päivityksen aikaan, jatketaan nauhoittamista edelleen. Jos liike on jotain muuta kuin käve-lyä tai juoksua, asetetaan sijainnin tarkkuus huonoimmaksi, sekä etäisyys-suodatin suurimmaksi mahdolliseksi resurssienkäytön optimoimiseksi. Jos käyttäjä liikkui jalan edellisen päivityksen aikaan, pysäytetään nauhoitus ja jos käyttäjä ei liikkunut, ei sovelluksen tarvitse reagoida.

Koska Apple toivoo taustasuoritustiloja käytettävän säästeliäästi ja vain silloin kun niille on oikea tarve, pitää myös sijaintitietojen käyttämiselle sovelluksessa olla jokin syy. Vaikka toimeksiantaja on kiinnostunut ainoastaan kiihtyvyysan-turista saatavasta datasta, eikä sijaintitietojen käyttämiselle ole oikeaa tar-vetta, vaan sen ainoa tarkoitus tässä sovelluksessa on pitää se taustalla auki, täytyi sille keksiä jotain muuta käyttöä, jotta sovellus vastaa Applen suuntavii-voja. Päädyttiin siis testaamaan, miten sijaintitietojen pohjalta saisi laskettua päivittäin kuljettua matkaa (kuva 15).

```
func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
    guard let currentLocation = locations.last else {
        return
    }
    if self.isMoving{
        if let lastLocation = lastLocation {
            let distance = lastLocation.distance(from: currentLocation)
            if distance > 50{

            } else {
                let currentDate = Date()

                cumulativeDistance += distance
                print("Distance traveled: \(distance) meters")
                iOSPlugin.updateCumulativeDistance(cumulativeDistance)

                if var lastResetDate = lastResetDate, !Calendar.current.isDate(lastResetDate, inSameDayAs: currentDate){
                    cumulativeDistance = 0.0
                    lastResetDate = currentDate
                }
            }
        }
        lastLocation = currentLocation
    }
}
```

Kuva 15. Päivän aikana kuljetun matkan päivittäminen kumulatiivisesti

Kuvassa 15 näkyvää Core Location -sovelluskehyksen CLLocationManager-Delegate-protokollan sisältämää locationManager(_:didUpdateLocations:)-instanssimetodia kutsutaan automaattisesti aina, kun käyttäjän sijainti päivittyy. (locationManager… s.a.) Kun uusi sijaintitieto saapuu, lasketaan nykyisen ja edellisen sijainnin välinen välimatka. Koska käyttäjän liikkuessa muuten kuin jalan, sijaintia päivitetään harvemmin ja pienemmällä tarkkuudella, voi kahden sijaintipäivityksen välinen etäisyys olla satojakin kilometrejä, eikä suurempia kuin 50 metrin etäisyyksiä kahden sijainnin välillä lasketa siksi mukaan päivän matkaan. Muussa tapauksessa päivän kumulatiivinen matka päivitetään ja tieto lähetetään lisäosan metodikanavan kautta Flutter-puolelle. Päivän matka lisättiin myös näkymään prototyypin käyttöliittymässä käyttäjälle. Kumulatiivinen matka nollautuu vuorokauden vaihtuessa.

Koska liikkeentunnistus ja kumulatiivisen matkan päivittäminen vaikutti toimivalta, implementoitiin ne toiseen testisovellukseen samalla tyylillä, ainoana poikkeuksena recordAccelerometer-metodin sijasta kiihtyvyysanturin nauhoitukseen käytettiin ensimmäisessäkin testisovelluksessa käytettyä CMMotionManager-luokkaa ja sen startAccelerometerUpdates-metodia. Nauhoituksen aloittamismetodi implementoitiin muuten samoin, kuin ensimmäisessä sovellusversiossa, mutta anturin nauhoittamisen sijaan aloitettiin liikkeentunnistus, joka nauhoitti anturia tarvittaessa kuvan 13 mukaisesti. Nauhoituksen lopettamismetodissa puolestaan lopetettiin lisäksi liikkeentunnistus. Toinen sovellusversio ei siis hirveästi poikennut ensimmäisestä, siihen vain lisättiin liikkeentunnistus ja kuljetun matkan laskeminen.

5.2 Testaus

Ensimmäinen viikko testattiin taas 100Hz nauhoitustaajuudella. Koska sovellus nauhoitti kiihtyvyysanturia nyt vain silloin, kun käyttäjä liikkuu, ei ollut enää varmuutta johtuiko suuret aukot datassa siitä, että käyttäjä ei ole liikkunut, vai siitä, että järjestelmä on sulkenut sovelluksen. Oman kokemuksen mukaan urheilu- ja älykellot, sekä esimerkiksi Oura-älysormus tuntuvat kehottavan jaloittelemaan noin tunnin välein, mutta koska tämä ei aina ole mahdollista, määriteltiin tässä vaiheessa suureksi aukoksi kaksi tuntia ilman dataa. Todennäköisempää toki on, että testaajat eivät vain ole liikkuneet näiden aukkojen aikana,

kuin että he olisivat kahden tunnin välein tarkistaneet nauhoittaako sovellus edelleen ja laittaneet takaisin päälle, jos se on sulkeutunut.

Keskimäärin ensimmäisen viikon aikana näitä suuria aukkoja oli jokaisella puhelimella kaksi tai kolme kappaletta. Selkeitä käyttöjärjestelmän aiheuttamia sovelluksen sulkeutumisia (esimerkiksi viimeiset datat ovat klo 18.24 illalla ja seuraava nauhoite alkaa seuraavana päivänä klo 14.39) oli ensimmäisen viikon aikana noin puolilla puhelimista, mutta ei kuitenkaan kaikilla joka päivä. Yleensä nämä tapahtuivat suht myöhään illalla tai aamuyön aikana. Ensimmäiseen sovellusversioon verrattuna käyttöjärjestelmä sulki sovellusta kuitenkin huomattavasti vähemmän. Kaikki puhelimet lähettivät ainakin kerran automaattisesti dataa ensimmäisellä viikolla ja noin puolet kaksi tai kolme kertaa viikon aikana. Toiseen sovellusversioon vaihdettaessa testaajat huomasivat sovelluksen akunkulutuksen laskevan huomattavasti.

Toisella testausviikolla nauhoitustaajuus laskettiin taas 50Hz. Toisen viikon aikana suuria aukkoja oli jokaisella puhelimella keskimäärin kaksi kappaletta päivässä, mutta toinen iPhone 6s ja iPhone 7 lähettivät dataa, jossa aukkoja oli enimmillään jopa viisi kappaletta päivän aikana. Toisen viikon aikana selkeät käyttöjärjestelmän aiheuttamat sovelluksen sulkeutumiset kasvoivat ja niitä oli viikon aikana jokaisella puhelimella vähintään kaksi kertaa. Tämäkin oli kuitenkin paljon vähemmän, kuin ensimmäisen sovellusversion kanssa. Toisen viikon aikana oli ehkä myös huomattavissa pientä laskua testaajien motivaatiossa, koska dataa puuttui useammaltakin puhelimelta enemmän kuin aiemmin. Näissä tapauksissa järjestelmä oli sulkenut sovelluksen ja sovelluksen toiminta oli tarkistettu vasta myöhään seuraavana päivänä tai jopa vasta kahden päivän päästä, eli sovelluksen tilaa ei tarkistettu enää yhtä usein kuin aiemmilla viikoilla. 100Hz versioon verrattuna 50Hz versiolla akunkulutus laski testaajien mukaan edelleen. Toisella viikolla iPhone 14 Pro oli ainut puhelin, joka ei lähettänyt dataa ollenkaan automaattisesti, mutta muuten tilanne pysyi suhteellisen samanlaisena, kuin edellisellä viikolla. Vain testikäytössä olevan iPhone SE:n toiminnassa ei vieläkään nähty eroa muihin puhelimiin verrattuna.

6 TULOKSET

Sovelluksen aktiivisuustila määrittelee, mitä sovellus voi milläkin hetkellä tehdä. Vain aktiivinen, passiivinen ja taustatilassa oleva sovellus voi suorittaa koodia ja muissa tiloissa ei tehdä mitään lukuun ottamatta muutamia Applen omia poikkeuksia. Koska sovelluksen normaaliin sykliin kuuluu, että se siirtyy taustalta keskeytetyksi hyvin pian sovelluksesta poistuttaessa, täytyy kiihtyvyysanturin taustalla nauhoittamista varten olla jokin keino, jolla sovelluksen saa pysymään taustatilassa. Taustalla koodin suorittamiseksi pidempään Apple tarjoaa taustasuoritustiloja.

Taustasuoritustiloja on useita erilaisia, mutta tässä työssä oltiin kiinnostuneita ainoastaan sijaintipäivitysten vastaanottamisesta ja taustatehtävistä vastaavista taustasuoritustiloista. Sijaintipäivitysten vastaanottaminen mahdollistaa sovelluksen pysymisen taustatilassa jatkuvasti, mikä antaa keinon kiihtyvyysanturin nauhoittamisen myös taustalla. Taustatehtävistä vastaavat taustasuoritustilat puolestaan antavat sovellukselle luvan suorittaa suhteellisen lyhytkestoisia tehtäviä taustalla.

Kiihtyvyysanturin nauhoittamiselle löytyy Core Motion -sovelluskehyksestä kaksi eri tapaa. recordAccelerometer-metodilla voi nauhoittaa maksimissaan 12 tuntia putkeen ennalta määritellyllä 50Hz taajuudella ja metodi toimii poikkeuksellisesti myös keskeytetty- ja ei käynnissä -tiloissa. CMMotionManager-luokan avulla puolestaan voidaan kerätä ja tallentaa dataa täysin reaaliajassa, mutta se toimii vain, kun sovellus on aktiivisena tai taustatilassa, jolloin tarvitaan jotain taustasuoritustilaa, joka pitää sovelluksen käynnissä myös taustalla. Tämän teorian pohjalta pystyttiin kehittämään sovellus, joka nauhoittaa puhelimen kiihtyvyysanturia taustalla.

Ensimmäisen sovellusversion testauksessa 100Hz ja 50Hz viikkojen välillä ei ollut huomattavaa eroa. Käyttöjärjestelmä sulki kaikissa puhelimissa sovelluksen yleisesti kerran tai kaksi päivässä, eikä välttämättä edes joka päivä. Vain iPhone 7 Plus nauhoitti toisella viikolla jopa kuusi vuorokautta putkeen. Lähes kaikki puhelimet kykenivät nauhoittamaan kiihtyvyysanturia koko sen ajan,

minkä järjestelmä antoi sovellukselle suoritusaikaa. Ensimmäisen viikon aikana muutamassa vanhimmassa puhelinmallissa oli pieniä katkoja nauhoituksissa, mutta toisen viikon aikana katkoja oli huomattavasti enemmän.

Toisen sovellusversion testauksessa 100Hz ja 50Hz viikkojen välillä ei myöskään ollut kovin suurta eroa. Suurten katkojen määrä päivän aikana pysyi suhteellisen samana, joskin toisen viikon aikana selkeiden käyttöjärjestelmän aiheuttamien sovelluksen sulkeutumisten määrä kasvoi, mutta niitäkään ei ollut päivittäin jokaisessa puhelimessa. Ensimmäiseen sovellusversioon verrattuna sovellus sulkeutui kummallakin viikolla kuitenkin huomattavasti vähemmän.

Nauhoitustaajuuden vaikutusta akunkulutukseen tutkittiin vaihtamalla molemmissa sovellusverioissa toisella viikolla puolet pienempään, 50Hz taajuuteen. Akun kulutus väheni viikko viikolta ja vaikka 100Hz taajuudella nauhoittaminen toisella sovellusversiolla vei jo huomattavasti vähemmän akkua kuin 50Hz nauhoittaminen ensimmäisellä sovellusversiolla, saatiin akunkäyttöä optimoitua edelleen laskemalla taajuudeksi 50Hz myös toisella sovellusversiolla. Ne puhelimet, jotka lähettivät dataa hyvin jo ensimmäisen sovellusversion aikana, jatkoivat edelleen datan lähetystä hyvin toisenkin version aikana, mutta muiden puhelinten datanlähetys parani huomattavasti, kun sovellusta pyrittiin muuten optimoimaan. Vaikka testausten aikana oli puhelimia, jotka suorittivat lähettämisestä vastaavan taustatehtävän useampanakin päivänä putkeen, ei yksikään puhelin kuitenkaan kyennyt suorittamaan lähetystä joka ikisenä päivänä huolimatta siitä, kuinka hyvin sovellus yritettiin optimoida resurssien käytön näkökulmasta, jotta käyttöjärjestelmä antaisi luvan tehtävän suorittamiselle.

7 JOHTOPÄÄTÖKSET

Tavoitteena oli tutkia, soveltuuko iPhone kiihtyvyysanturin reaaliaikaiseen nauhoittamiseen taustalla ja pyrkiä toteuttamaan toimeksiantajalle kaksi eri versiota tällaisesta sovelluksesta. Ensimmäisen version tarkoituksena oli tarkastella, kykeneekö iPhone nauhoittamaan kiihtyvyysanturia katkeamatta vuorokauden ympäri. Toisen version tarkoituksena oli toteuttaa toiminto, joka nauhoittaisi kiihtyvyysanturia vain silloin, kun käyttäjä liikkuu, jotta voidaan minimoida turhan datan määrä ja optimoida sovelluksen resurssien käyttöä.

Ensimmäisen sovellusversion toisen testausviikon aikana oli hetkiä, jolloin käyttöjärjestelmä ei ollut sulkenut jotain puhelimia joka päivä, huolimatta siitä kuinka paljon sovellus vei esimerkiksi akkua. Tästä voidaan siis päätellä, että iPhone kykenee kyllä nauhoittamaan kiihtyvyysanturia vuorokauden ympäri katkeamatta, kun olosuhteet ovat käyttöjärjestelmän mielestä oikeanlaiset. Testauksen aikana iPhone 7 Plus oli ainut puhelin, joka nauhoitti kiihtyvyysanturia kuusi vuorokautta putkeen, mutta se oli myös yksi niistä puhelimista, joiden lähettämässä datassa oli paljon pieniä katkoja ilman, että käyttöjärjestelmä oli sulkenut sovelluksen. Tämän perusteella voidaan sanoa, että vaikka iPhone kykenee toisinaan nauhoittamaan kiihtyvyysanturia ympäri vuorokauden – tai useammankin – ei dataa välttämättä saa täysin ilman katkoja, eikä tässä työssä toteutettu lähestymistapa kiihtyvyysanturin nauhoitukselle kuitenkaan sovellu pitkäaikaiseen jatkuvaan nauhoitukseen, mikä toimeksiantajalla olisi tavoitteena. Tavoite vuorokauden ympäri kiihtyvyysanturia nauhoittavasta sovelluksesta toteutui siis kuitenkin jollain tasolla.

Toinen sovellusversio onnistui myös osaltaan. Sovellus saatiin nauhoittamaan kiihtyvyysanturia vain silloin, kun käyttäjä liikkuu jalan. Myös sovelluksen resurssien käyttöä saatiin optimoitua. Optimoinnin puolesta puhuu, ei vain akunkäytön väheneminen, vaan myös se, että käyttöjärjestelmältä saatiin lupa suorittaa taustatehtäviä enemmän kuin ensimmäisen version kanssa, ja se, että käyttöjärjestelmä ei sulkenut sovellusta enää niin usein. Optimointia ei kuitenkaan pystytty toteuttamaan niin hyvin, että käyttöjärjestelmä ei puuttuisi ollenkaan sovelluksen toimintaan ja se toimisi taustalla täysin luotettavasti.

Näiden tietojen valossa lopputuloksena voidaan pitää sitä, että iPhone soveltuu kyllä puhelimen oman kiihtyvyysanturin nauhoittamiseen, mutta ei pitkäaikaisesti taustalla ainakaan tässä työssä toteutetuin metodein. Sovelluksen optimointi niin, että käyttöjärjestelmä ei sulje sovellusta ja antaa luvan taustatehtävien suorittamiseen on haastavaa, eikä täysin toimeksiantajan haluamalla tavalla toimivaa sovellusta saatu toteutettua ja aihe vaatii lisää tutkimusta.

8 POHDINTA

Työ onnistui suhteellisen hyvin ottaen huomioon sen, että iOS on järjestelmän itsenäisyyden vuoksi alustana haastava halutun tyyppisen sovelluksen toteuttamiseen. Molemmat sovellusversiot saatiin kuitenkin toteutettua jollakin tasolla, vaikka ne eivät täysin toimineetkaan niin kuin oli suunniteltu. Työ lisäsi myös toimeksiantajan ymmärrystä siitä, millainen iOS järjestelmänä on ja mitä sovellusta kehittäessä täytyy ottaa huomioon. Lisäksi työ hieman myös herätteli toimeksiantajaa siihen, millaisia kompromisseja tulevaisuudessa täytyy mahdollisesti tehdä kaupallisen tuotteen kehittämisen kanssa.

Applen dokumentaatioon tutustuminen oli välttämätöntä, jotta löydettiin oikeat sovelluskehykset sovelluksen toteuttamiseen, vaikkakaan se ei aina auttanut itse toteutuksessa. Paljon meni aikaa siihen, että tietoa piti etsiä myös muualta ja itse kokeilla, mitä eri sovelluskehysten komponenteilla pystyy tekemään. Lisäksi dokumentaatio avasi myös paljon sitä, miten iOS järjestelmänä toimii. ja auttoi hahmottamaan, mikä on ylipäätään kehittäjänä mahdollista ja mikä ei. Toki teorian pohjalta haasteita oli odotettavissa nauhoituksen päällä pysymisen ja taustatehtävien osalta, koska käyttöjärjestelmän suorittamaan optimointiin ei voi kehittäjänä vaikuttaa, mutta en olettanut sen olevan niin haastavaa, kuin miksi se lopulta osoittautui.

Tulevaisuudessa voisi kokeilla, antaako käyttöjärjestelmä taustatehtäville enemmän suoritusaikaa, jos ne tunnin sijaan ajastettaisi esimerkiksi kuuden tunnin välein. On hyvin mahdollista, että järjestelmä eväsi tehtävän suorituksen, koska sitä pyydettiin suoritettavaksi aivan liian usein ja harvemmin pyydettynä olisi lupa voitu saada helpommin. Lisäksi heräsi ajatuksia siitä, että recordAccelerometer-metodia kokeiltaisi 12 tunnin nauhoitukseen uudestaan, mutta nauhoituksen uusimiseen käytettäisi jotain muuta kuin taustatehtäviä. Mielessä kävi nauhoituksen aloittaminen merkittävien sijaintimuutosten avulla, jolloin nauhoitus alkaisi, kun käyttäjä on liikkunut 500 metriä esimerkiksi aamulenkin tai työmatkan aikana. Ongelmana on kuitenkin se, että kaikki data ensimmäisiltä 500 metriltä jää nauhoittamatta ja jos käyttäjä tekeekin aamulenkin juoksumatolla, ei sijainti muutu ollenkaan, jolloin kaikki liikedata jää nauhoittamatta. Yksi vaihtoehto voisi myös olla ajastetut push-ilmoitukset, mutta järjestelmä rajoittaa myös niiden lähettämistä, vaikkei tarkkaa tietoa ole

siitä, rajoittaako se niitä yhtä paljon kuin taustatehtävien suorittamista. Palvelimelta tulevia ilmoituksia ei rajoiteta kuitenkaan yhtä paljon kuin sovellukselta itseltään tulevia, mutta tässäkin voi olla riskinä se, että niillekään ei anneta lupaa tarpeeksi usein.

Tällä hetkellä sovelluksen toteuttaminen halutunlaisesti vaikuttaa hyvin epävarmalta, ja vaikka mieleen tulee useampiakin tapoja, joita voisi vielä kokeilla, on niidenkin kanssa edessä omat ongelmansa. Lisäksi on varmasti olemassa vielä muitakin keinoja, joita ei ole osattu ajatella tai ne eivät ole vielä tulleet vastaan. Sovelluksen jatkokehitys vaatiikin siis vielä uusiin sovelluskehyksiin tutustumista ja niiden testailua, ja työtä on edessä vielä suhteellisen paljon, kun kehitys jatkuu edelleen kohti kaupallista tuotetta.

LÄHTEET

AltexSoft. 2021. The Good and the Bad of Swift Programming Language. Blogi. Saatavissa: https://www.altexsoft.com/blog/the-good-and-the-bad-of-swift-programming-language/ [viitattu 31.1.2024].

Acc.elerometer s.a. App Store. WWW-dokumentti. Saatavissa: https://apps.apple.com/fi/app/acc-elerometer/id1631020465?l=fi [viitattu 10.12.2023].

AccelMeter s.a. App Store. WWW-dokumentti. Saatavissa: https://apps.apple.com/fi/app/accelmeter/id346525189?l=fi [viitattu 10.12.2023].

Anahertz – Frequency Analysis s.a. App Store. WWW-dokumentti. Saatavissa: https://apps.apple.com/fi/app/anahertz-frequency-analysis/id1310201833?l=fi [viitattu 10.12.2023].

Apple Developer Videos. 2020. Background execution demystified. Videoleike. Saatavissa: https://developer.apple.com/videos/play/wwdc2020/10063/ [viitattu 3.11.2023].

Background Tasks s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/backgroundtasks [viitattu 3.11.2023].

CMMotionActivityManager s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/coremotion/cmmotionactivitymanager [viitattu 29.11.2023].

CMMotionManager s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/coremotion/cmmotionmanager [viitattu 6.2.2024].

Configuring background execution modes s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/xcode/configuring-background-execution-modes [viitattu 3.11.2023].

Core Location s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/corelocation [viitattu 29.11.2023].

Core Motion s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/coremotion [viitattu 29.11.2023].

distanceFilter s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/corelocation/cllocationmanager/1423500-distancefilter [viitattu 7.2.2024].

Ekren, E. 2023. What Is Xcode and How to Use It? Blogi. Saatavissa: https://www.netguru.com/blog/what-is-xcode-and-how-to-use-it [viitattu 31.1.2024].

FAQ s.a. Flutter. WWW-dokumentti. Saatavissa: https://docs.flutter.dev/resources/faq [viitattu 31.1.2024].

Getting raw accelerometer events s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/coremotion/getting_raw_accelerometer_events [viitattu 3.11.2023].

Grouios, G., Ziagkas, E., Loukovitis, A., Chatzinikolaou, K. & Koidou, E. 2022. Accelerometers in Our Pocket: Does Smartphone Accelerometer Technology Provide Accurate Data? *Sensors* 23, 192. WWW-dokumentti. Saatavissa: https://doi.org/10.3390/s23010192 [viitattu 29.11.2023].

Handling location updates in the background s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/corelocation/handling_location_updates_in_the_background [viitattu 6.2.2024].

Kananen, J. 2017. Kehittämistutkimus interventiotutkimuksen muotona: Opas opinnäytetyön ja pro gradun kirjoittajalle. Jyväskylä: Jyväskylän ammattikorkeakoulu. [viitattu 14.11.2023].

Kirvan, P. 2022. What are GPS (global positioning system) coordinates? WWW-dokumentti. Päivitetty 08.2022. Saatavissa: https://www.techtarget.com/whatis/definition/GPS-coordinates [viitattu 29.11.2023].

locationManager(_:didUpdateLocations:) s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/corelocation/cllocationmanagerdelegate/1423615-locationmanager [viitattu 23.3.2024].

Managing your app's life cycle s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle [viitattu 29.11.2023].

Manohar, C., McCrady, S., Fujiki, Y., Pavlidis, I. & Levine, A. 2011. Evaluattion of the Accuracy of a Triaxial Accelerometer Embedded into a Cell Phone Platform for Measuring Physical Activity. *Journal of Obesity & Weight Loss Therapy* 1, 106. WWW-dokumentti. Saatavissa: https://doi.org/10.4172/2165-7904.1000106 [viitattu 29.11.2023].

pausesLocationUpdatesAutomatically s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/corelocation/cllocationmanager/1620553-pauseslocationupdatesautomatical [viitattu 7.2.2024].

Perera, C. 2019. The execution states of an iOS application. Blogi. WWW-dokumentti. Saatavissa: https://medium.com/@chinthaka01/the-execution-states-of-an-ios-application-84e117132e27 [viitattu 29.11.2023].

Physics Toolbox Accelerometer s.a. App Store. WWW-dokumentti. Saatavissa: https://apps.apple.com/fi/app/physics-toolbox-accelerometer/id1008160133?l=fi [viitattu 10.12.2023].

Polar Beat: Juoksu & Fitness s.a. App Store. WWW-dokumentti. Saatavissa: https://apps.apple.com/fi/app/polar-beat-juoksu-fitness/id555252645?l=fi [viitattu 10.12.2023]. .

recordAccelerometer(forDuration:) s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/coremotion/cmsensorrecorder/1615987-recordaccelerometer [viitattu 3.11.2023].

Ridjanovic, D & Balbaert, I. 2013. Learning Dart. Birmingham: Pact Publishing. E-kirja. Saatavissa: https://ebookcentral.proquest.com/lib/xamk-ebooks/reader.action?docID=1441767 [viitattu 23.1.2024].

Sharma, S. 2020. What Is Accelerometer? How to Use Accelerometer in Mobile Devices? Blogi. Saatavissa: https://www.credencys.com/blog/accelerometer/#:~:text=The%20accelerometer%20is%20an%20in,uses%20the%20value%20of%20XYZ [viitattu 29.11.2023].

Sparkfun s.a. Accelerometer Basics. WWW-dokumentti. Saatavissa: https://learn.sparkfun.com/tutorials/accelerometer-basics/all [viitattu 29.11.2023].

Speedometer∞ s.a. App Store. WWW-dokumentti. Saatavissa: https://apps.apple.com/fi/app/speedometer/id1114655078?l=fi [viitattu 10.12.2023].

Sports Tracker for All Sports s.a. App Store. WWW-dokumentti. Saatavissa: https://apps.apple.com/fi/app/sports-tracker-for-all-sports/id426684873?l=fi [viitattu 10.12.2023].

Strava: Run, Bike, Hike s.a. App Store. WWW-dokumentti. Saatavissa: https://apps.apple.com/fi/app/strava-run-bike-hike/id426826309?l=fi [viitattu 10.12.2023].

Supported deployment platforms s.a. Flutter. WWW-dokumentti. Saatavissa: https://docs.flutter.dev/reference/supported-platforms [viitattu 31.1.2024].

Suunto s.a. App Store. WWW-dokumentti. Saatavissa: https://apps.apple.com/fi/app/suunto/id1230327951?l=fi [viitattu 10.12.2023].

Using background tasks to update your app s.a. Apple Developer Documentation. WWW-dokumentti. Saatavissa: https://developer.apple.com/documentation/uikit/app_and_environment/scenes/preparing_your_ui_to_run_in_the_background/using_background_tasks_to_update_your_app [viitattu 6.2.2024].