Tomi Fjäder

Sovelluskehitys React Nativella

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

11.11.2016

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Tomi Fjäder Sovelluskehitys React Nativella 37 sivua + 2 liitettä 11.11.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander

Insinöörityössä tutustuttiin mobiilisovelluskehityksen toteutustapoihin, hybridisovellusten ohjelmistokehyksiin sekä React Native -ohjelmistokehykseen. Työn tavoitteena oli oppia React Nativen käyttöä ja kehittää sovellus, jolla on täysin yhteinen koodikanta Android- ja iOS-alustoille.

React Native on ohjelmistokehys, jonka avulla voi kehittää hybridisovelluksia Android-, iOS- ja Windows Phone-alustoille. Sen toiminta pohjautuu alustakohtaisten natiivikomponenttien käsittelyyn, mikä poikkeaa perinteisistä hybridisovelluksista. Perinteiset hybridisovellukset toteutetaan web-tekniikoilla, ja ne hyödyntävät kohdealustan WebView-säiliötä. React Nativen avulla kehitetyt sovellukset ovat alustariippumattomia natiivisovelluksia.

Työn teoriaosuudessa syvennyttiin React Nativen toimintaperiaatteisiin ja pyrittiin havainnollistamaan muun muassa renderöinnin toimintaa. Käytännön osuudessa kuvataan sovelluskehitystä React Nativella oman sovelluksen ja koodiesimerkkien avulla.

Lopputuloksena syntyi tavoitteiden mukainen Stats Collector -mobiilisovellus. Sovelluksen avulla voi kerätä tilastoja jalkapallo-ottelusta. Sovellusta on tarkoitus kehittää lisää ja julkaista se Googlen ja Applen sovelluskaupoissa.

Avainsanat	React Native, sovellus, Android, iOS, ohjelmistokehys

Author(s) Title Number of Pages Date	Tomi Fjäder Application Development with React Native 37 pages + 2 appendices 11 November 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Simo Silander, Senior Lecturer

This thesis focuses on software development with React Native. Different methods of mobile application development and hybrid frameworks are also covered. The goal of the thesis was to learn to develop applications with the React Native framework and to develop a hybrid application with a common codebase for Android and iOS.

React Native is a framework for developing Android, iOS and Windows Phone applications. It uses native components directly which differs from traditional hybrid frameworks. Traditional hybrid applications are developed using web-technologies and they use WebView containers to display the content. Applications made with React Native are platform independent native apps.

The theory section concentrates on the operational principles of React Native. The section tries to illustrate for example how the rendering works in React Native. The software development with React Native is also described, including code samples and screenshots from Stats Collector application.

The Stats Collector application was the result of the project and it met all the objectives. The application is for collecting match statistics from a football match. The application is supposed to be released in Google's and Apple's app stores after some further development.

Keywords	React Native, application, Android, iOS, framework

Sisällys

Lyhenteet

IDE	Integrated Development Environment. Ohjelmointiympäristö sovellusten kehittämiseen.
NDK	Native Development Kit. Työkalu, joka mahdollistaa C- ja C++-kielen käytön natiivi Android-kehityksessä.
OS	Operating System. Käyttöjärjestelmä.
MVC	Model-View-Controller. Ohjelmistoarkkitehtuurityyli.
MVVM	Model-View-ViewModel. Ohjelmistoarkkitehtuurityyli.
SDK	Software Development Kit. Työkalu, jonka avulla voi kehittää sovelluksia tietyille kohdealustoille.
DOM	Data Object Model. Dokumenttioliomalli, joka kuvaa rakenteisen dokumentin puuna.
UWP	Universal Windows Platform. Yleiskäyttöinen Windows-alusta.
XML	Extensible Markup Language. Yläkäsite merkintäkielille.
GCD	Grand Central Dispatch. Työkalu tehtävien suorittamiseen.
CLI	Command Line Interface. Komentorivikäyttöliittymä.
npm	Node Package Manager. Työkalu pakettien hallintaan.
FPS	Frames Per Second. Näytölle piirrettyjen kuvien määrä sekunnissa.
CSS	Cascade Style Sheet. Kieli dokumentin tyylien kuvaamiseksi.
DPI	Dots Per Inch. Tuuman sisältämä pisteiden lukumäärä, mikä kuvastaa näytön tarkkuutta.

1 Johdanto

Mobiilisovellukset ovat hyvä keino tavoittaa asiakkaita ja mainostaa palveluita. Yhä useammat yritykset ovat havahtuneet tarvitsevansa oman sovelluksen. Pankit kehittävät sovelluksia maksamiseen ja kaupat bonuksien hallintaan. Sovellusten lukumäärä on kasvanut runsaasti viimeisen kuuden vuoden aikana. Liitteen 1 kuvista nähdään, että App Storen sovellusten lukumäärä on yli 13-kertainen ja Google Playn 80-kertainen vuoteen 2010 verrattuna.

Sovellusten kehittäminen Applen, Androidin ja Microsoftin puhelimille on yksilöllistä. Kullakin alustalla on omanlainen käyttöjärjestelmä, tyyliohjeet, julkaisuprosessi, ohjelmointikieli ja ohjelmointiympäristö. Mikäli kehitettävä sovellus aiotaan julkaista usealle alustalle samanaikaisesti, tulee kehittäjiä olla vähintään yhtä monta kuin tuettavia alustoja. Scrum-viitekehys (3, s. 6) suosittelee kehittäjäryhmän kooksi 3-9 henkilöä. Ohjelmistokehyksiä käyttämällä kehittäjäryhmän kokoa voi pienentää ilman, että sen tuottavuus kärsii. Ohjelmistokehyksillä samoja toiminallisuuksia ei tarvitse ohjelmoida useaan kertaan, vaan yhdellä ohjelmointikielellä voidaan kattaa useampi alusta, jolloin kehitystyö on kustannustehokkaampaa.

Insinöörityön aiheena on tutustua React Native -nimiseen ohjelmistokehykseen, sekä verrata sitä Androidin ja iOS:n natiivitoteutuksiin. Työn tavoitteena on kehittää mobiilisovellus, jolla on yhteinen koodikanta Android- ja iOS-laitteille. Työssä tutustutaan sovellusten toteutustapoihin, verrataan tarjolla olevia ohjelmistokehyksiä sekä syvennytään React Nativen toimintaan. Luvut 2,3 ja 4 esittelevät tutkimustyön tuloksia ja ovat teoriapainotteisia. Luku 5 on käytännönläheinen katsaus sovelluskehitykseen React Nativella, mikä pitää sisällään koodiesimerkkejä ja huomioita insinöörityön kehitystyöstä.

2 Sovelluksen toteutustavat

Kuten perinteisiä tietokoneohjelmia, myös puhelinsovelluksia voidaan toteuttaa eri tavoin. Tietokoneohjelmat voivat olla käyttöjärjestelmälle asennettavia ohjelmia tai tietokoneen selaimessa toimivia ohjelmia. Vastaavasti myös puhelinsovellusten toteutustavat on jaettu eri kategorioihin. Sekä Vuorisen että Haapahovin (4; 5) mukaan toteutustavat voi jakaa kolmeen kategoriaan: natiivi-, hybridi- ja web-sovelluksiin. Kehittyneet ohjelmistokehykset muodostavat neljännen kategorian: modernit hybridisovellukset. Kaikilla toteutustavoilla on hyviä ja huonoja puolia mutta jokaiselle löytyy oma käyttökohde.

Ennen toteutustavan valitsemista täytyy ymmärtää sovelluksen kehitystyön vaatimuksia. Valintaan on monta vaikuttavaa tekijää, kuten henkilöresurssit, projektin aikataulu, käyttöympäristö, laitteistovaatimukset sekä vaaditut toiminnallisuudet. Vaatimusten ymmärtämisen jälkeen voidaan valita sopiva toteutustapa.

2.1 Natiivisovellus

Natiivisovelluksilla tarkoitetaan yksittäiselle mobiilialustalle ohjelmoitua sovellusta, kuten Vuorinen (4) kuvailee. Kohdealusta määrittää kehittämiseen käytettävän ohjelmointikielen ja ympäristön. Natiivisovellus voi olla esimerkiksi Android-, iOS- tai Windows-sovellus.

Tavallisesti Android-sovellukset ohjelmoidaan Java-ohjelmointikielellä käyttäen Android Studio IDE:tä. Ohjelmointiympäristönä voi käyttää muitakin vaihtoehtoja, mutta Android Studio on Googlen virallinen IDE Androidille. Googlen Android NDK:n ansiosta sovelluskehityksessä voi käyttää myös C- ja C++-kieliä. Android-sovellusten kehittämistä ei ole rajoitettu tietokoneen käyttöjärjestelmän mukaan. Tarvittavat työkalut saa asennettua sekä Windows- että Unix-käyttöjärjestelmiin, kuten Linuxiin ja Mac OS:ään.

Toisin kuin Androidilla iOS-sovelluskehitys on sidottu yhteen käyttöjärjestelmään. Ohjelmointiympäristö on Applen Xcode IDE, joka on saatavilla ainoastaan OS X -käyttöjärjestelmille. Ohjelmointikielinä toimivat Objective-C ja Swift. Molemmat kielet toimivat myös yhdessä, joten kehittäjät voivat hyödyntää molempia projekteissaan.

Microsoftin Windows Phone -käyttöjärjestelmälle on myös oma kieli ja ohjelmointiympäristö. Käyttöjärjestelmän alhaisen markkinaosuuden takia kyseessä ei ole merkittävä tekijä mobiilimarkkinoilla (taulukko 1), jonka vuoksi insinöörityössä keskitytään Android- ja iOS-sovelluksiin.

Taulukko 1. Maailmanlaajuinen älypuhelimien myyntitilasto käyttöjärjestelmittäin (6).

Käyttöjärjestelmä	**2016 kvartaali 1 myydyt yksiköt (tuhansia)**	**2016 kvartaali 1 Markkinaosuus (%)**	**2015 kvartaali 1 myydyt yksiköt (tuhansia)**	**2015 kvartaali 1 Markkinaosuus (%)**
Android	293 771,2	84,1	264 941,9	78,8
iOS	51 629,5	14,8	60 177,2	17,9
Windows	2 399,7	0,7	8 270,8	2,5
Blackberry	659,9	0,2	1 325,4	0,4
Muut	791,1	0,2	1 582,5	0,5
Yhteensä	**349 251,4**	**100,0**	**336 297,8**	**100,0**

Koska natiivisovellus ohjelmoidaan käyttöjärjestelmäkohtaisesti, saadaan laitteen suorituskyky hyödynnettyä optimaalisesti. Juuri suorituskyky on yksi natiivisovellusten suurimmista hyödyistä. Käyttöjärjestelmäkohtaisen ohjelmoinnin myötä sovelluksessa voidaan hyödyntää puhelimen laitteistoa ja sensoreita. Sen ansiosta myös sovelluksen ulkoasu ja toiminnallisuus vastaavat käyttäjän tottumuksia. Koska sovelluksen käyttämä data voidaan tallentaa puhelimen muistiin, ei sovelluksen käyttö vaadi internetyhteyttä. Kehittäjä voi julkaista natiivisovelluksia Googlen ja Applen virallisissa sovelluskaupoissa.

Natiivisovelluksella on myös huonoja puolia, kuten koodin huono siirrettävyys alustalta toiselle. Kehittäjä ei pysty hyödyntämään Android-sovelluksen Java-koodia, mikäli hän haluaa toteuttaa saman sovelluksen myös iOS-laitteille. Ohjelmointityö täytyy aloittaa alusta uudella kielellä, uudella ohjelmointiympäristöllä ja mahdollisesti eri käyttöjärjestelmällä. Samojen toteutusten kirjoittaminen uudelleen tekee natiivisovelluksen kehittämisestä kallista ja aikaavievää. Viralliset sovelluskaupat eivät varmista, että käyttäjä asentaa sovelluksen viimeisimmän version, mikä vaikeuttaa vianetsintää.

2.2 Web-sovellus

Web-sovellukset ovat internetsivuja tai verkkosovelluksia, jotka toimivat puhelimen internetselaimessa. Toimintatapa on samanlainen, kuin perinteisellä pöytäkoneella käytettäessä. Näkymän skaalautuvuus mobiililaitteille toteutetaan tyyliasetuksilla, jotka vaihtelevat laitteen näytön koon mukaan. (5.)

Tämän hetken suosituimpia tapoja toteuttaa web-sovellus ovat Angular-ohjelmistokehys ja React JavaScript -kirjasto. Suosio johtuu osittain JavaScript-ohjelmointikielestä, johon molemmat toteutustavat pohjautuvat. Angular tarjoaa kehyksen, jonka avulla sovelluksen saa helposti kehitettyä MVC- tai MVVM-mallin mukaisesti. Reactin toimintaan perehdytään tarkemmin luvussa 4.

Koska web-sovellukset toimivat selaimessa, ei niillä ole pääsyä laitteen muihin ominaisuuksiin, kuten kamera ja anturit. Mikäli sovelluksen tarvitsee hyödyntää puhelimen ominaisuuksia, ei web-sovellus ole sopiva vaihtoehto. Web-sovelluksen kehittämisessä on samoja ongelmia kuin verkkosivujen toteuttamisessa: sovellus ei toimi ilman internetyhteyttä ja kaikkien selainten tukeminen vaatii yksilöllisiä sääntöjä. Verkkoteknologioilla toteutettu käyttöliittymä ei vastaa puhelimen natiivia käyttöliittymää, jolloin se ei vastaa käyttäjän tottumuksia. Web-sovellusten suorituskyky ei yletä natiivisovellusten tasolle, eikä niitä voi jakaa virallisissa sovelluskaupoissa.

Web-sovellusten vahvuuksiin lukeutuu koodin siirrettävyys kaikkiin käyttöympäristöihin. Kehittäjän tarvitsee ohjelmoida sovellus kerran, jonka jälkeen se toimii samalla tavalla tietokoneella, tabletilla ja puhelimella. Vuorisen ja Haapahovin (4; 5) mukaan web-sovellukset ovat myös edullisempia kehittää kuin natiivisovellukset. Kehityskustannuksiin vaikuttaa myös potentiaalisten kehittäjien määrä, sillä web-kehittäjiä on enemmän tarjolla kuin Android- tai iOS-osaajia. Käyttäjillä on aina käytössä sama versio sovelluksesta, mikä helpottaa vianetsintää.

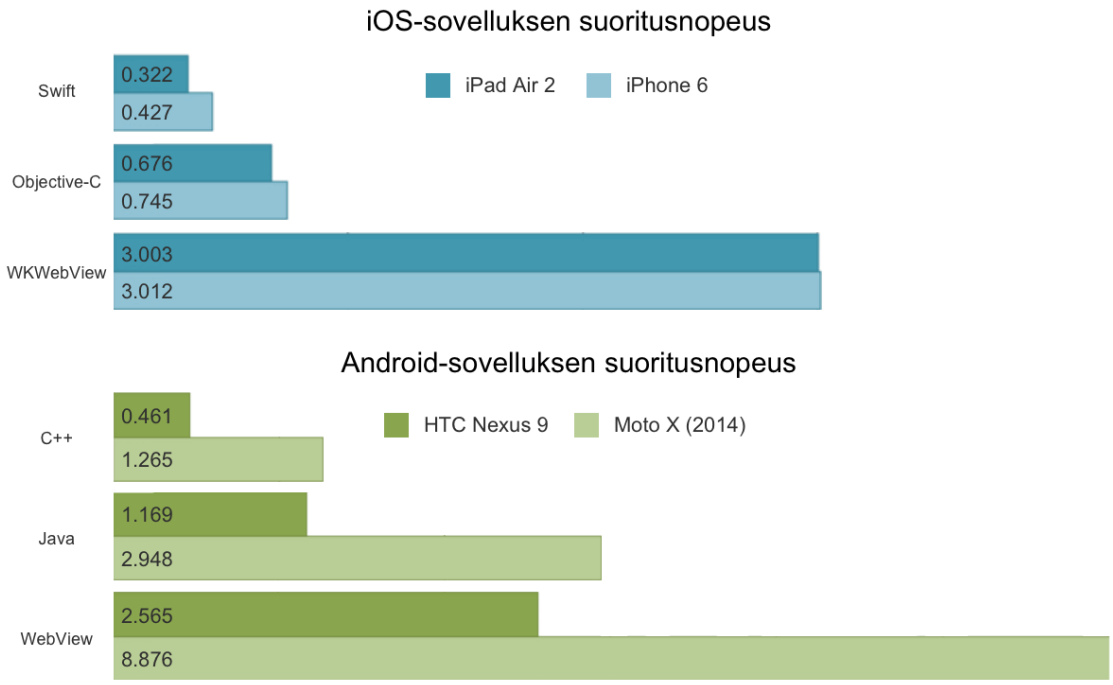
2.3 Hybridisovellus

Savuojan (7, s. 18) mukaan hybridisovellukset muodostavat yhteen paketoidun kokonaisuuden, joka pitää sisällään alustariippumattoman web-sovelluksen sekä natiivin

WebView-säiliön. Säiliö toimii puhelimen selaimena, joka peittää sille tavanomaiset tunnusmerkit kuten osoiterivin, jolloin käyttäjä ei huomaa eroa natiivisovellukseen. Savuoja (7, s. 20) toteaa, että ilman natiiveja käyttöliittymäkomponentteja kehittäjä helposti unohtaa käyttöjärjestelmän tyyliohjeet, jolloin hybridisovelluksen käyttöliittymä poikkeaa natiivisovelluksista.

Hyvin toteutettu hybridisovellus yhdistää web-sovelluksen sekä natiivisovelluksen parhaita puolia. Kehityskustannukset ovat natiivisovellusta alhaisemmat. Sovelluksella on pääsy tiettyihin puhelimen ominaisuuksiin, ja käyttöliittymä on natiivisovelluksen kaltainen. (8.) Hybridisovelluksen hyviin puoliin lukeutuvat myös koodin siirrettävyys kaikille alustoille sekä virallisten sovelluskauppojen hyödyntäminen.

Hybridisovellusten huonoja puolia ovat muun muassa kauppapaikkojen sovellusten lukumäärä ja huono suorituskyky. Suorituskyky ei yllä natiivisovellusten tasolle ja vaikuttaa sovelluksen käytettävyyteen. Kuva 1 havainnollistaa natiivisovellusten ja hybridisovellusten suorituskyvyn eroa.

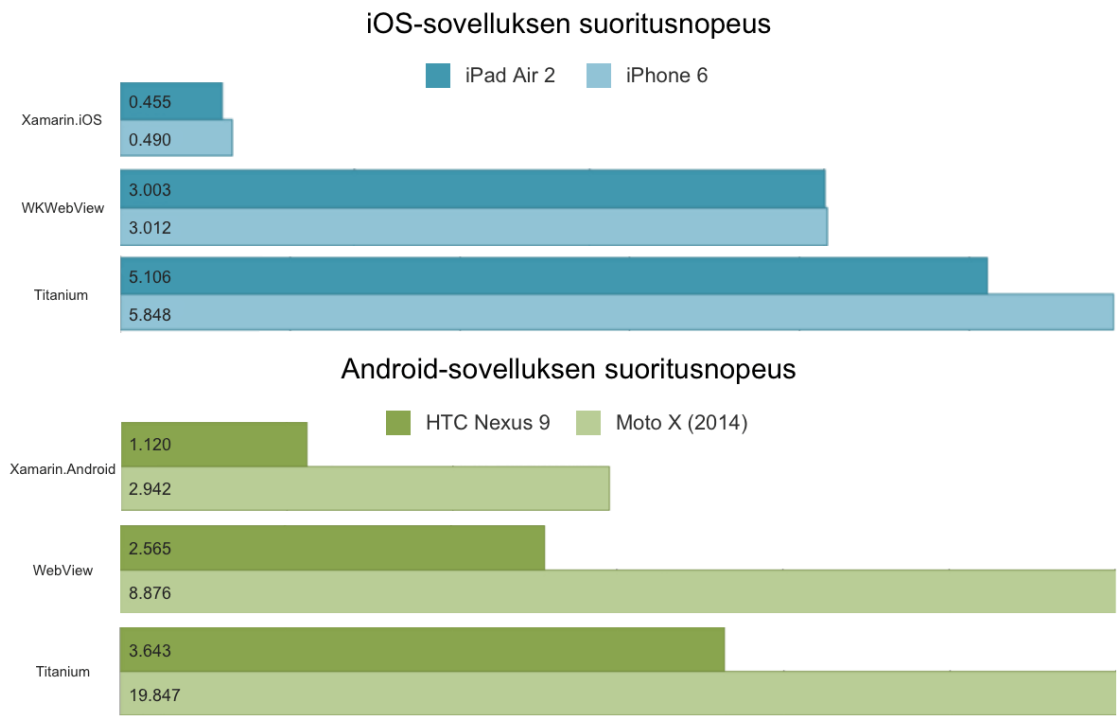


Kuva 1. Cheungin suorittaman, laitteiston ja ohjelmointikielen laskennallista suorituskykyä mittaavan, testin tulokset. Tulokset on esitetty sekunteina, ja pienempi arvo kuvastaa parempaa suorituskykyä. (9.)

3 Ohjelmistokehykset

Tässä työssä ohjelmistokehyksellä tarkoitetaan mobiilisovellusten kehittämiseen käytettäviä ohjelmistokehyksiä. Rusilan (10, s. 9) mukaan ohjelmistokehykset voidaan jakaa kahteen luokkaan: web-pohjaiset ja natiivit. Web-pohjaisten ohjelmistokehysten avulla tuotetaan perinteisiä hybridisovelluksia, jotka pohjautuvat laitteiston WebView-säiliöihin. Web-pohjaisia ohjelmistokehyksiä ovat muun muassa Ionic, Apache Cordova ja Titanium.

Natiivien ohjelmistokehysten avulla sovellus toteutetaan ilman WebView-säiliötä tai muutakaan selaimeen pohjautuvaa menetelmää. Tämän vuoksi natiiveilla ohjelmistokehyksillä tuotettuja sovelluksia ei voi pitää perinteisinä hybridisovelluksina. Natiiveja ohjelmistokehyksiä ovat esimerkiksi Xamarin ja React Native. Kuvan 2 mittaustulosten perusteella voidaan todeta, että natiivilla ja web-pohjaisella ohjelmistokehyksellä tuotetun sovelluksen suorituskyvyn ero on merkittävä.



Kuva 2. Cheungin suorittaman, laitteiston ja ohjelmointikielen laskennallista suorituskykyä mittaavan, testin tulokset. Tulokset on esitetty sekunteina ja pienempi arvo kuvastaa parempaa suorituskykyä. Xamarin.iOS ja Xamarin.Android kuvaavat natiivilla ohjelmistokehyksellä kehitetyn sovelluksen suorituskykyä. Titanium ja WebView kuvaavat web-pohjaisella ohjelmistokehyksellä kehitetyn sovelluksen suorituskykyä. (9.)

3.1 Xamarin

Xamarin on vuonna 2011 perustettu yritys, jonka Microsoft osti maaliskuussa 2016. Yritysoston myötä Microsoftin Visual Studio IDE pitää nykyään sisällään Xamarin-työkalut. Applen tietokoneilla työskenteleville on asennettavissa Xamarin Studio IDE. Visualin ja Xamarin Studion ohjelmointikielenä on käytettävissä joko C# tai F# ja sillä voi kehittää sovelluksia Android-, iOS- ja Windows-alustoille. (11.)

Friedman (11) ilmoittaa kirjoituksessaan aikomuksista julkaista Xamarin SDK:n lähdekoodi MIT-lisenssin alaisena. Nykyään edellä mainittu aikomus on realisoitunut. Tätä ennen suljettu ympäristö ja kirjastojen tuen puute olivat suuria Xamarinin heikkouksia. Huhtikuussa 2015 Xamarinia oli ladattu yli miljoona kertaa, mikä on merkki sen suuresta suosiosta (12). On odotettavissa, että Microsoftin yhteisön myötä Xamarinin kehittäjien määrä tulee kasvamaan entisestään.

Xamarinin etuja ovat sen hyvä suorituskyky ja natiivikomponenttien tuki, mutta onko se paras vaihtoehto alustariippumattomaan sovelluskehitykseen. Suorituskyvyn lisäksi alustariippumattoman sovelluksen yksi tärkeimpiä ominaisuuksia on koodin siirrettävyys alustojen välillä. Xamarinin sivuston mukaan Xamarin.iOS- ja Xamarin.Android-tuotteilla saavutetaan keskimäärin 75-prosenttisesti ja Xamarin.Forms-tuotteella lähes 100-prosenttisesti yhteinen koodikanta eri alustoille. Xamarin.Forms-tuote sopii käytettäväksi, jos sovellus vaatii vain vähän alustakohtaisia toimintoja ja koodin alustojen välinen siirrettävyys on tärkeämpää kuin yksilöllinen käyttöliittymä. Knighton (13) mainitsee artikkelissaan, että Xamarin.Formsin avulla voi saavuttaa korkeintaan 90-prosenttisesti ja muilla tuotteilla vain 30-prosenttisesti yhteisen koodikannan.

3.2 React Native

React Nativen lähdekoodi julkaistiin maaliskuussa 2015, joten kyseessä on todella uusi ohjelmistokehys. React Native on Facebookin tuote, joka pohjautuu Facebookin ja Instagramin kehittämään React JavaScript -kirjastoon. Reactille ominaista on sovellusten jakaminen yksilöllisiin komponentteihin, joista kukin toimii näkymänä. Lisäksi virtuaalisen DOM:n ansiosta React päivittää ainoastaan muuttuneet komponentit koko DOM:n sijaan. (14.) React Native pyrkii tuomaan vastaavanlaisen toiminnallisuuden mobiilisovellusten

kehittämiseen. Virtuaalisen DOM:n hyödyntämistä käsitellään tarkemmin luvussa 4 ja React Nativen komponentteja luvussa 5.

Ensimmäisen julkaisun aikaan React Native tuki vain iOS-käyttöliittymää. Syyskuussa 2015 myös Android tuki julkaistiin (15). iOS-lähdekoodin julkistaminen ennen Android-versiota aiheutti sen, että osa komponenteista oli käytettävissä vain iOS-alustalle. Android-julkaisun jälkeen komponentteja on muokattu toimivaksi molemmilla alustoilla. Sittemmin React Nativen yhteisön jäsenet ovat kehittäneet tuen myös Windows UWP:lle.

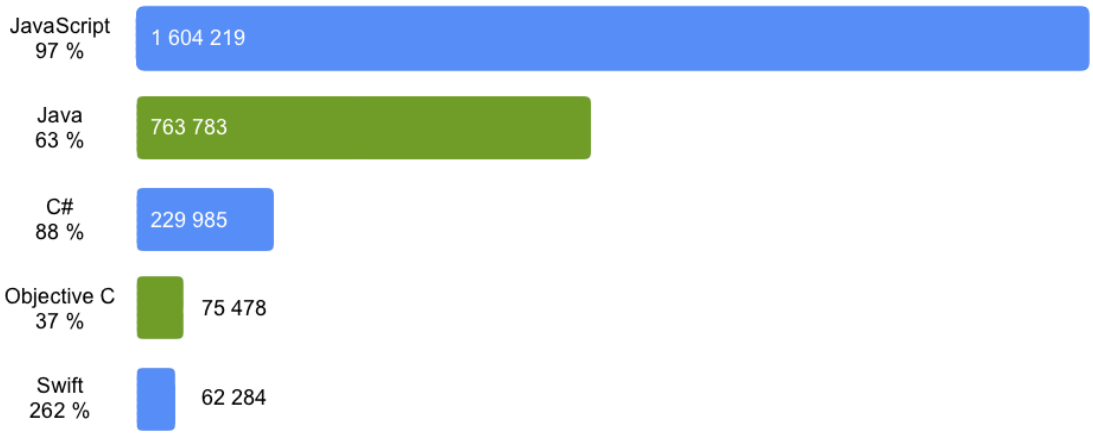
Koska kyseessä on uusi ohjelmistokehys, sen kehittyminen on todella nopeaa. Versiopäivitykset tapahtuvat noin kahden viikon sykleissä, mikä on syytä ottaa huomioon kehitystyössä. Insinöörityön teon aikana React Native ehti päivittyä versiosta 0.33 versioon 0.37. Nopea kehittyminen on osittain aktiivisen yhteisön ansiota, joka tuottaa React Nativelle liitännäisiä ja komponentteja.

React Nativella ei ole tiettyä ohjelmointiympäristöä, vaan kehittäjä voi käyttää haluamaansa tekstieditoria. Facebook on kehittänyt sovelluskehitystä helpottamaan Nuclide-työkalun, joka on asennettavissa Atom-tekstieditorin liitännäisenä. React Native -projekteja voi hallita myös sen päälle rakennetulla Exponent-työkalulla. React Native tukee kolmansien osapuolien kehittämiä työkaluja, joten Exponentin kaltaisten sovellusten voi odottaa lisääntyvän.

React Nativella on runsaasti hyviä puolia, minkä vuoksi se on saavuttanut lyhyessä ajassa suosiota kehittäjien keskuudessa. Laborde (16) kirjoittaa, ettei ole koskaan nähnyt juniorikehittäjien saavuttaneen vastaavaa tuottavuutta niin lyhyessä ajassa. Insinöörityön sovelluskehityksessä havaittuja hyötyjä olivat:

- Kehittäjän ei tarvitse koota sovellusta nähdäkseen muutokset.
- Kehittäjä voi hyödyntää tuttuja Chrome-kehitystyökaluja.
- Kehittäjä voi hyödyntää JavaScript-kirjastoja, mikäli ne eivät ole DOM-riippuvaisia.
- Kehittäjän tarvitsee kirjoittaa koodi vain yhden kerran.
- Kehittäjä voi käyttää natiivia ohjelmointikieltä tarvittaessa.

Xamarinia houkuttelevamman React Nativesta tekee JavaScript, joka on kehityksessä käytettävä ohjelmointikieli. Apple mahdollistaa JavaScript-muutosten päivittämisen sovellukseen verkon yli ilman App Storen tarkastusprosessia (17, s. 20). JavaScriptin lisäksi kehityksessä käytetään JSX-kieltä, joka on XML-tyyppinen määrittelykieli. Kuva 3 osoittaa, kuinka paljon suositumpi kieli JavaScript on kuin Xamarinin käyttämä C# tai natiivit ohjelmointikielet.



Kuva 3. GitHubin suosituimmat ohjelmointikielet avattujen Pull Requestien perusteella mitattuna. Prosenttiarvo kuvaa kasvua edelliseen mittaustulokseen verrattuna. (18.)

Vaikka React Native on vielä kehityksen alla, niin monet yritykset käyttävät sitä tuotekehityksessään. Sovellukset kuten Facebook, Facebook Ads Manager, Facebook Groups, Instagram ja Airbnb on kehitetty React Nativea hyödyntäen. (19.) RND Works (20) tuotekehitystalon blogissa mainitaan, että heidän kehittämä Veikkauksen mobiilisovellus on myös kehitetty React Nativella.

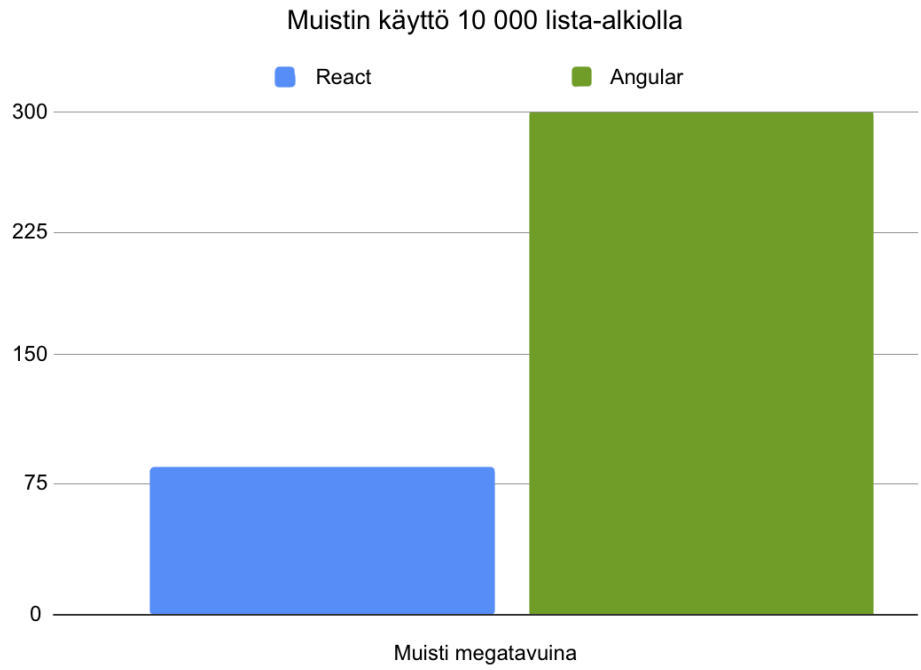
4 React Nativen toiminta

Kuten työssä aiemmin mainittiin, React Nativen toiminta pohjautuu React JavaScript -kirjastoon. Yhtäläisyyksien ansiosta Reactia käyttävien web-kehittäjien on helppo siirtyä kehittämään mobiilisovelluksia. Tässä luvussa käydään läpi Reactin toimintaperiaatteita, jotka pätevät myös React Nativeen sekä React Nativen ominaisuuksia.

4.1 React

React on käyttöliittymien rakentamiseen kehitetty JavaScript-kirjasto. Kyseessä ei ole MVC-mallia hyödyntävä ohjelmistokehys kuten Angular, vaan Reactia pidetään MVC-mallin V-komponenttina, joka huolehtii käyttöliittymän esittämisestä. Käytännössä tämä tarkoittaa sitä, että datan hallintaan kannattaa hyödyntää ulkopuolista kirjastoa.

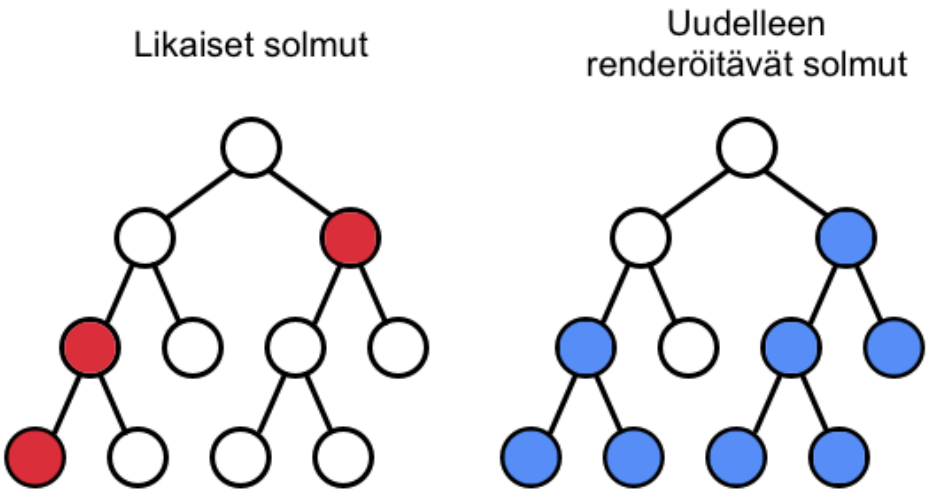
Reactin yksi tärkeimpiä ominaisuuksia on web-sovelluksen hyvän suorituskyvyn ylläpitäminen. Suorituskyvyn mittareita ovat muun muassa sovelluksen käyttämän muistin määrä sekä näytettävän sisällön renderöintiin käytetty aika. Tässä työssä renderöinnillä tarkoitetaan komponenttien piirtämistä päätelaitteen näytölle. Kuvasta 4 nähdään, että Reactilla kehitetyn web-sovelluksen käyttämä muistin määrä on huomattavasti pienempi kuin vastaava Angularilla toteutettuna.



Kuva 4. Yksinkertaisen sovelluksen käyttämä muistin määrä Reactilla ja Angularilla toteutettuna. Sovellus näyttää listalla 10 000 komponenttia. Käytetyn muistin määrä on esitetty megatavuina. (21.)

4.1.1 Renderöinti

DOM:n manipulointi ja sisällön uudelleenrenderöinti on raskas toimenpide, jolla on heikentävä vaikutus sovelluksen suorituskykyyn. Sovellusten toiminnan kannalta on tärkeää, että käyttäjällä on nähtävissä reaaliaikaista dataa, mikä vaatii toistuvaa uudelleen renderöintiä. Reactia käytettäessä kyseinen ongelma on ratkaistu virtuaalisen DOM:n avulla. (17, s. 26.) Kyseessä on Chedeaun (22) mukaan joukko JavaScript-objekteja, jotka kuvaavat varsinaista dokumenttioliomallia. Kuva 5 esittää, kuinka Reactilla pyritään virtuaalisen DOM:n avulla minimoimaan uudelleen renderöitävien objektien määrä ja päivitetään varsinaiseen dokumenttioliomalliin mahdollisimman pieni muutos.



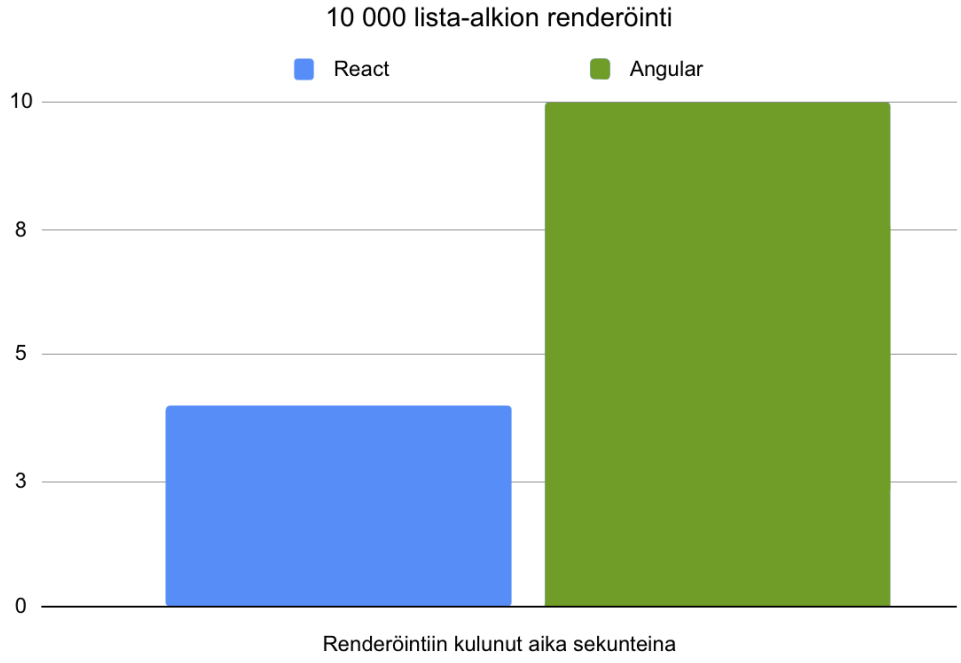
Kuva 5. Chedeaun (22) esittämä kuva virtuaalisen DOM:n toiminnasta. Reactin algoritmi merkitsee virtuaalisen dokumenttioliomallin muuttuneet solmukohdat likaisiksi. Likaiset solmut ja niiden jälkeläiset renderöidään uudelleen. Kuvassa likaiset solmut on merkitty punaisella ja uudelleen renderöitävät solmut sinisellä.

Reactin renderöinnillä on olemassa tietty elinkaari. Kun web-sivu on renderöity, siihen kytketään React-komponentteja. Tämän jälkeen renderöinti kohdistuu React-komponentteihin koko dokumenttioliomallin sijasta. Komponentteja renderöidään uudelleen, mikäli niiden tila tai ominaisuudet muuttuvat. Virtuaalisessa dokumenttioliomallissa erilaisuuksien havaitsemisalgoritmi merkitsee muuttuneet komponentit likaisiksi. Likaiset solmut ja niiden jälkeläiset renderöidään uudelleen. (17, s. 28.) Komponentit sekä niiden ominaisuudet ja tilat esitellään työn luvussa 5.

Eisenmanin (17, s. 26) mukaan monet kehittäjät pitävät Reactin virtuaalista dokumenttioliomallia ainoastaan suorituskyvyn optimoinnin työkaluna, mutta sen abstraktion avulla React Native voi käyttää sitä natiivien mobiilikomponenttien renderöinnissä. Koska natiivien mobiilikomponenttien renderöinti poikkeaa runsaasti dokumenttioliomallista, React Nativen toiminta edellyttää alustakohtaisen sillan hyödyntämistä. Sillan avulla sovelluksen JavaScript-säie kommunikoi asynkronisesti pääsäikeen kanssa.

4.1.2 Algoritmi

Perinteisesti kahden puurakenteen erilaisuuksien havaitsemisalgoritmi on ollut kertaluokkaa O(n^3). Reactin kehittäjät onnistuivat heuristiikkojen avulla parantamaan algoritmin vaativuutta. Muutosten myötä algoritmin vaativuus on kertaluokkaa O(n). (23.) Kuva 6 osoittaa, että Reactin tehokkaan algoritmin ja keveyden ansiosta, sillä kehitetty sovellus on huomattavasti Angularilla kehitettyä sovellusta nopeampi.



Kuva 6. Yksinkertaisen sovelluksen renderöintiin käyttämä aika Reactilla ja Angularilla toteutettuna. Sovellus näyttää listalla 10 000 komponenttia. Renderöintiin käytetty aika on kuvattu sekunteina. (21.)

Sovelluksen rakennetta suunniteltaessa kannattaa ottaa huomioon, että käyttöliittymä on jaettu tarpeeksi yksinkertaisiin komponentteihin. Lisäksi muuttuvat komponentit kannat-

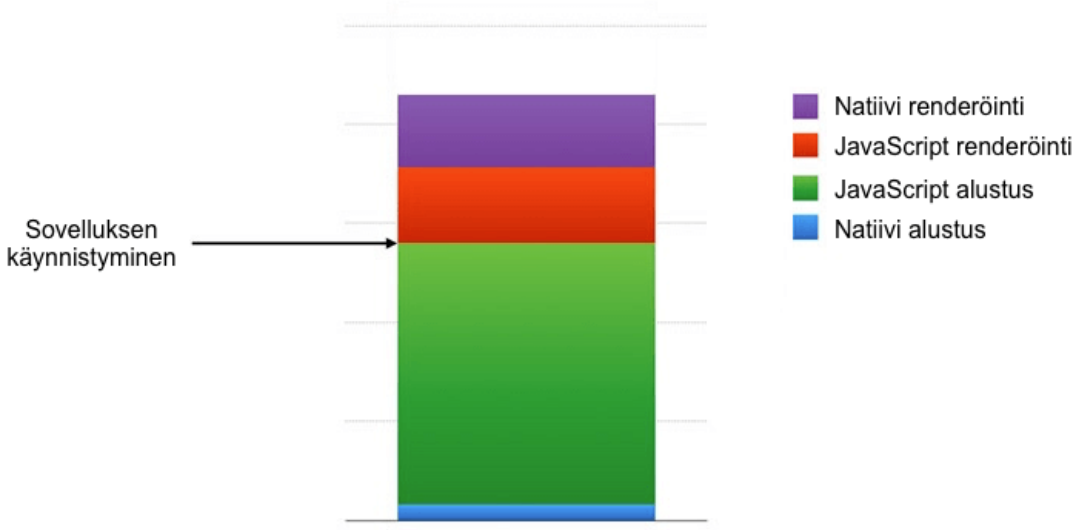
taa sijoittaa mahdollisimman alhaiselle tasolle dokumenttioliomallissa, jotta niiden päivityksellä olisi mahdollisimman pieni vaikutus muuhun sovellukseen. Reactin muutosten tarkistusalgoritmille on mahdollista ilmoittaa, että jotain solmua ei tarvitse päivittää, vaikka sen yllä oleva solmu olisi likainen.

4.2 Suorituskyky

Yksi React Nativen merkittävimmistä ominaisuuksista on sen hyvä suorituskyky. Merkittävästi perinteisiä hybridisovelluksia parempi suorituskyky on saavutettu ohjelmistokehyksen rakenteen avulla. React Nativen rakenne koostuu kahdesta säikeestä ja GCD-jonoista. GCD-jono on työkalu koodin suorittamiseen. Niiden avulla korvataan erillisten säikeiden tarve, minkä lisäksi ne ovat yksinkertaisempia käyttää ja tehokkaampia kuin koodin suorittaminen erillisissä säikeissä. (24.) React Nativessa hyödynnetään erityisesti GCD-jonojen asynkronista toimintaa.

Sovelluksen käynnistyessä tapahtuu paljon asioita ja React Nativen kehittäjät ovat tehneet runsaasti töitä käynnistykseen käytetyn ajan optimoimiseksi. De Baets ja kumppanit pystyivät puolittamaan Facebook-sovelluksen tapahtumat-näkymän latausajan. Valtaosa suorituskykyyn vaikuttavista muutoksista tehtiin ohjelmistokehykseen, joten kaikki React Nativen käyttäjät hyötyvät parannuksista. (25.)

Kuvasta 7 nähdään, mitä kaikkea tapahtuu React Nativella kehitetyn sovelluksen käynnistyessä. Sinisellä merkityllä alueella sovellus alustaa JavaScript virtuaalikoneen sekä kaikki natiivimoduulit, kuten muun muassa välimuisti ja käyttöliittymän ohjain. Vihreällä merkityllä alueella sovellus lukee JavaScript-paketin ja lataa sen virtuaalikoneelle. Virtuaalikone käsittelee paketin ja generoi tarvittavan tavukoodin. Punaisella merkityllä kohdalla sovellus luo React-komponenttien ilmentymät ja välittää ne käyttöliittymän ohjaimelle. Violetilla merkityllä alueella sovellus saa shadow-jonolta renderöitävien komponenttien koot, luo tarvittavat natiivikomponentit sekä sijoittelee ne pääsäikeessä. (25.)



Kuva 7. Facebook-sovelluksen tapahtumat-näkymän käynnistymisen tehokkuutta kuvaava diagrammi. Käynnistymiseen kuluva aika on jaettu neljään osa-alueeseen. (25.)

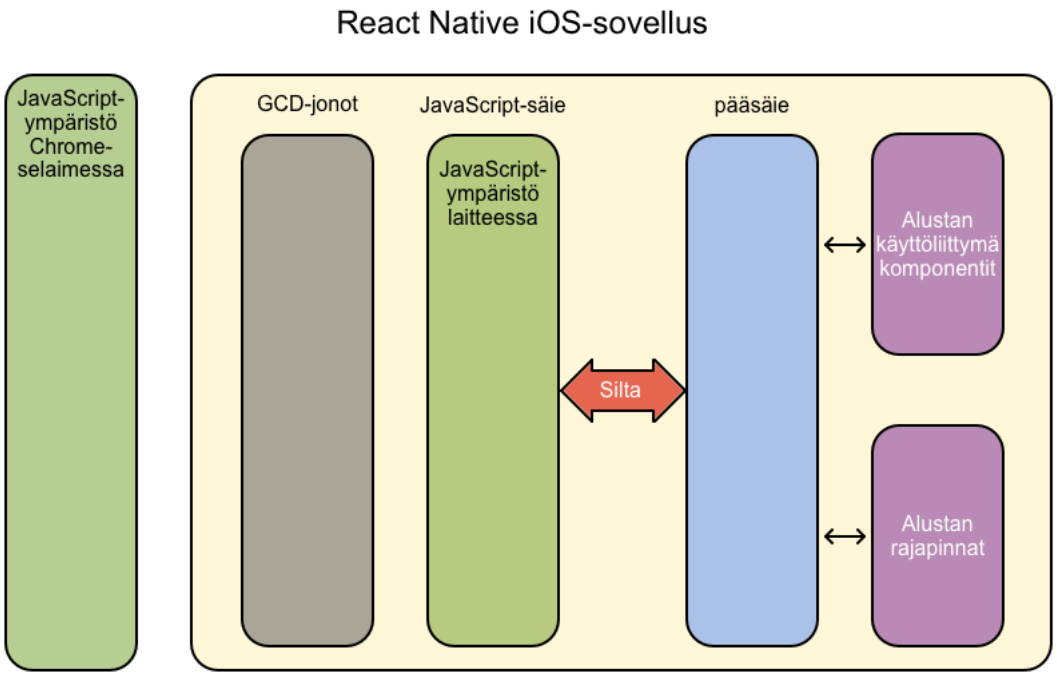
4.3 Säikeet

React Nativen säikeet ovat pääsäie sekä JavaScript-säie. Pääsäikeessä käsitellään natiivimoduulien renderöinti sekä suorituskyvyn kannalta raskaat toimenpiteet, kuten animaatiot. JavaScript-säikeessä kehittäjä määrittelee, miltä sovelluksen tulisi näyttää. Pääsäikeen pyhittäminen alustakohtaisten natiivimoduulien käsittelyyn mahdollistaa React Nativen hyödyntämisen usealla eri alustalla.

Kehittäjät toimivat pääsääntöisesti JavaScript-säikeessä, jossa muun muassa luodaan React-komponentteja. Koska React Nativen tarjoamat komponentit eivät täysin kata kaikkia natiivimoduuleja, on mahdollista, että kehittäjä joutuu luomaan komponentin itse. Tällöin kehittäjä luo rajapinnat JavaScript- ja pääsäikeeseen.

GCD-jonot on mahdollista mieltää yhdeksi säikeeksi, vaikka kyseessä ei ole perinteinen säie. Shadow-jono pitää huolta komponenttien asettelusta sekä käyttöjärjestelmän ulkoasusta. Se laskee renderöitävien natiivimoduulien koot asetettujen tyylien mukaisesti. (25.) Zagallon (26) mukaan jokaisella natiivimoduulilla on oma GCD-jono, jollei toisin ole määritelty.

Koska säikeet eivät voi suoraan kommunikoida keskenään, niiden väliin tarvitaan asynkroninen silta. Sekä JavaScript-säikeen että pääsäikeen tulee pystyä suorittamaan koodia estymättömästi. Tämän vuoksi asynkronisuus on välttämätöntä, jotta käyttöliittymä toimii ilman viiveitä. Kuva 8 havainnollistaa React Nativen arkkitehtuuria, jossa silta on tärkeässä roolissa.



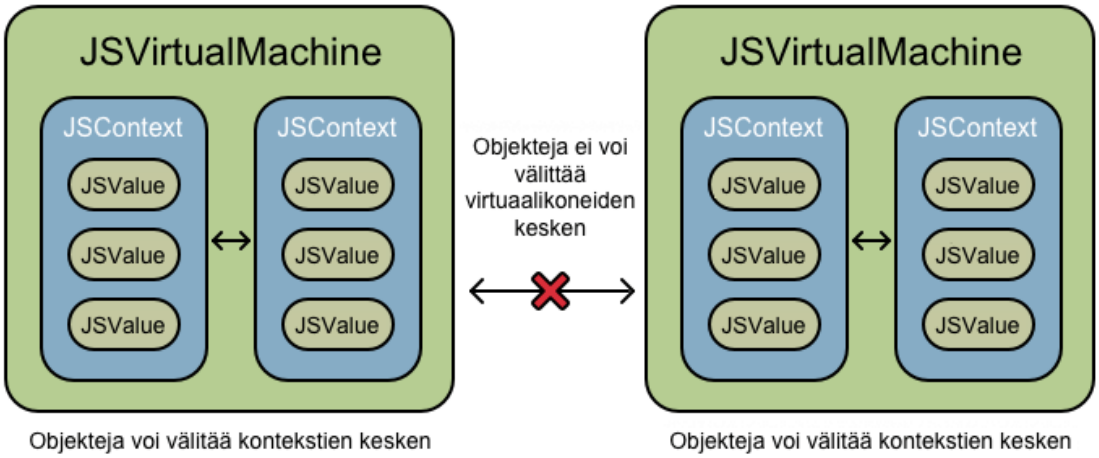
Kuva 8. Korkean tason kuvaus React Nativella kehitetyn sovelluksen arkkitehtuurista. JavaScript-ympäristö toimii tavallisesti sovelluksen omassa säikeessä mutta debugatessa se voi toimia myös Chrome-selaimessa. (27.)

4.4 JavaScript-ympäristö

JavaScriptillä on React Nativessa kaksi mahdollista toimintaympäristöä. Chrome-selaimella debugatessa JavaScript-koodit suoritetaan itse selaimessa. Kommunikointi natiivipuolen kanssa tapahtuu WebSocket-protokollan avulla. Tällöin sovellus hyödyntää Chromen omaa V8 JavaScript -moottoria. Simulaattoria tai mobiililaitetta käytettäessä sovellus käyttää JavaScriptCorea, joka on muun muassa Safarin käyttämä JavaScript-moottori. (28.)

Tarkemmin kuvailtuna JavaScriptCore on ohjelmistokehys, joka tarjoaa rajapinnan Applen WebKit JavaScript -moottoriin. Veszan (29) oppaassa kerrotaan, että JavaScriptCoren tärkeimpiä komponentteja ovat JSVirtualMachine, JSContext ja JSValue. JavaScript-koodi suoritetaan JSVirtualMachinen-luokan tarjoamassa virtuaalikoneessa. JSContext kuvastaa JavaScript-koodin ajoympäristöä. Kyseessä on yksi objekti, jonka vastine web-kehityksessä on ikkunaobjekti. JSValue-objekti kuvastaa mitä tahansa JavaScriptissä olevaa muuttujaa.

Kuvasta 9 nähdään, että JSContext-objektien välillä voi liikkua ainoastaa JSValue-objekteja ja virtuaalikoneiden välinen kommunikointi ei ole mahdollista. JavaScript-koodin samanaikainen suoritus täytyy toteuttaa usealla JSVirtualMachine-objektilla, koska usean säikeen käyttäminen ei ole JSVirtualMachine-objektissa mahdollista.



Kuva 9. Veszan (29) esittämä kuvaus JavaScriptCoren objektien välisestä kommunikoinnista.

5 Sovelluskehitys React Nativella

Tässä luvussa tutustutaan sovelluskehitykseen React Nativella ja verrataan sitä natiivikehitykseen. Osana insinöörityötä ohjelmoitiin Stats Collector -mobiilisovellus, jonka avulla voi kerätä jalkapallo-ottelun tilastoja. Sovelluksella tulee pystyä keräämään tilastoja monipuolisesti ja katsomaan kerättyä dataa. Tavoitteena oli pyrkiä 100-prosenttisesti yhteiseen koodikantaan. Yhteisellä koodikannalla tarkoitetaan sitä, että sovellus kääntyy Android- ja iOS-mobiilialustoille käyttäen samaa lähdekoodia.

5.1 Projektin hallinta

Natiivikehityksessä kehittäjä asentaa ohjelmointiympäristön kohdealustan perusteella. Android-sovellusten ympäristönä toimii Android Studio ja iOS:n ympäristönä Xcode. Alustariippumattomuus edellyttää molempien ohjelmointiympäristöjen asentamista sekä Mac-tietokoneen käyttöä. Mikäli sovellus toteutetaan vain Android-alustalle voi React Nativen asentaa Mac-, Windows- tai Linux-tietokoneelle, jolloin kehittäjän täytyy asentaa Android Studio. Jos sovellus toteutetaan vain iOS-alustalle, kehittäjän täytyy käyttää Mac-tietokonetta ja asentaa Xcode. Työssä keskitytään React Nativen käyttöön Mac-tietokoneilla, koska työn tavoitteena oli toteuttaa alustariippumaton sovellus.

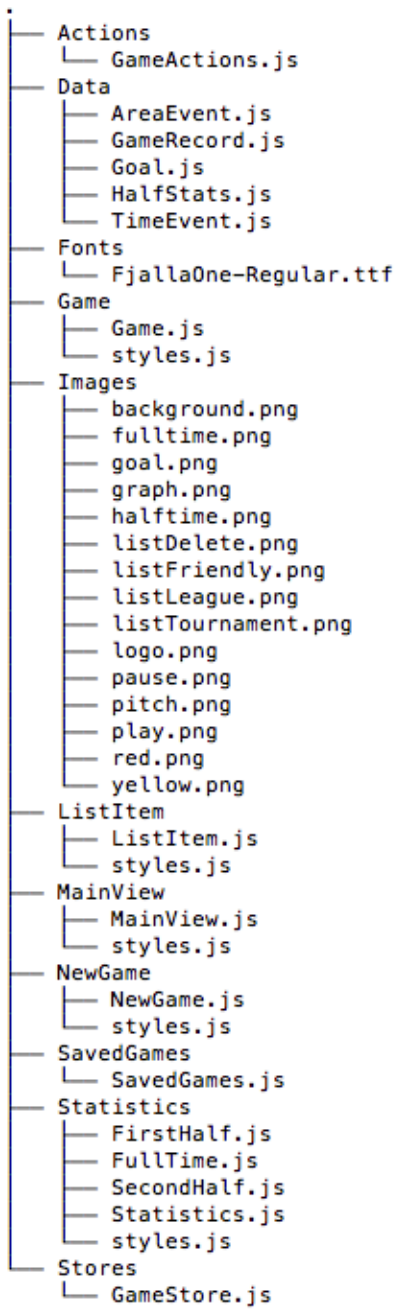
Kehitystä aloittaessa molempien ohjelmointiympäristöjen asentaminen vie aikaa, mutta kyseinen toimenpide tarvitsee tehdä vain yhden kerran. React Nativen ohjesivustolla (30) on hyvin dokumentoitu, kuinka ensimmäisen projektin saa luotua. Projektin luonnissa edettiin ohjesivuston mukaisesti ja asennettiin Node.js, Watchman ja React Native CLI. Node.js on ajonaikainen suoritusympäristö JavaScriptille, ja Watchman-työkalu seuraa muutoksia tiedostoissa.

React Native -projektin luominen on huomattavasti helpompaa kuin Android Studio- tai Xcode-projektien. Xcode-projektin luominen vaatii kaksi askelta sen jälkeen, kun käyttäjä on valinnut haluavansa luoda uuden projektin. Android Studiolla vastaava askelten määrä on vähintään kolme. React Nativella vaaditaan vain yksi komentorivikomento, ”react-native init StatsCollector”, kohdehakemistossa.

Uudessa projektissa React Native luo oletusmallisen hakemistorakenteen. Kuvasta 10 nähdään kyseinen rakenne, johon on lisätty readme-tiedosto sekä src-kansio. Projektin android-kansiossa on Android-projekti, jonka voi avata myös Android Studiolla. Vastaavasti ios-kansio sisältää Xcode-projektin, jonka voi avata Xcode-ohjelmointiympäristöllä. Index.android.js- ja index.ios.js-tiedostot ovat JavaScript-tiedostoja, jotka toimivat sovelluksen ensimmäisenä näkymänä. Kooditiedostot voi nimetä alustan mukaisesti kuvan 10 osoittamalla tavalla, jolloin ne käännetään ainoastaan nimetyn alustan sovellukselle.

Kuva 10. Stats Collector -projektin hakemistorakenne.

React Native -projektien rakenteen hallintaan ei ole vielä vakiintunutta mallia. Kehittäjän taustat vaikuttavat käytettävään rakenteeseen. Web-kehittäjät pyrkivät luonnostaan sijoittamaan tiedostot web-kehityksestä tuttuun tapaan, kun natiivikehittäjät hyödyntävät aiemmin käyttämistään ohjelmointiympäristöistä tuttuja rakenteita. Insinöörityön sovelluskehityksessä käytettiin kuvien 10 ja 11 kaltaista rakennetta. Komponentit ja niiden tyylit sijoitettiin omiin hakemistoihin, kuten myös fontit ja kuvat.



Kuva 11. Stats Collector -sovelluksen src-hakemiston rakenne.

Kuvan 10 package.json-tiedosto sisältää sovelluksen riippuvuudet, kuten käytetyt kirjastot ja kolmansien osapuolien kehittämät komponentit. Riippuvuuksien lisääminen ja poistaminen tapahtuvat npm:n avulla. Npm:n avulla asennetut riippuvuudet tallentuvat node_modules-hakemistoon.

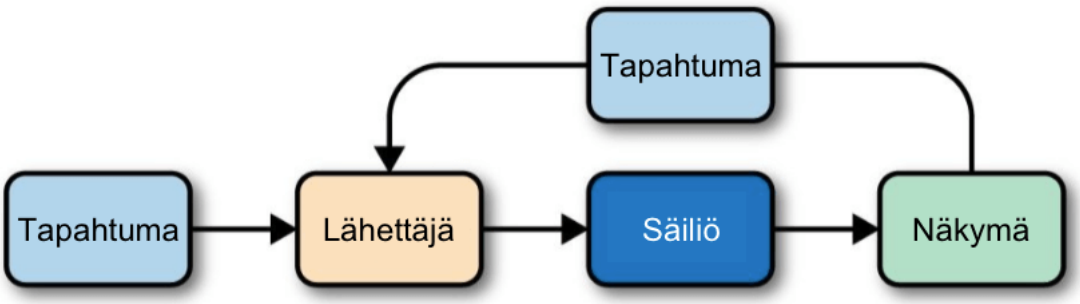
Sovelluksen käynnistäminen React Nativella on todella yksinkertaista. Projektin juurihakemistossa suoritetaan yksi komentorivikomento, ”react-native run-android” tai ”react-

native run-ios”, jonka jälkeen React Native jakaa JavaScript-tiedostot paikallisesti ja käynnistää emulaattorin. Sovellusta ei tarvitse koota uudelleen, jotta toteutetut muutokset näkyvät emulaattorissa tai puhelimessa. Tämä on suuri etu natiivikehitykseen verrattuna, sillä Xcodella sovellus tarvitsee koota uudelleen jokaisen muutoksen jälkeen, mikä vie turhaan aikaa. Android Studiolla on olemassa Instant Run-ominaisuus, joka siirtää vain muutetun koodin emulaattoriin tai puhelimeen. Instant Run-ominaisuus ei ole kuitenkaan yhtä nopea vaihtoehto kuin React Nativen tarjoamat työkalut.

5.2 Data

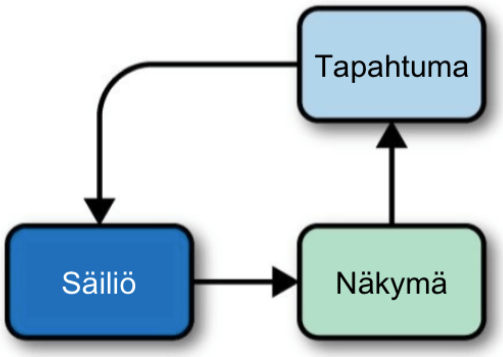
Stats Collector -sovelluksella voi kerätä paljon dataa, mikä edellyttää runsaasti muuttujia. Muuttujia ei tahdottu sijoittaa samaan tiedostoon, jossa määritellään näkymä, koska se rikkoisi MVC-ajattelumallia. Datan keräämisen vaatima toiminnallisuus siirrettiin erillisiin tiedostoihin Reflux-arkkitehtuurin mukaisesti. React Nativessa data kulkee yhdensuuntaisesti äitikomponentilta lapsikomponentille. Mikäli kehittäjä haluaa datan kulkevan myös toiseen suuntaan, täytyy hänen hyödyntää takaisinkutsuja. Takaisinkutsujen runsas käyttö tekee koodista vaikeasti luettavaa ja virhealtista.

Reflux pohjautuu Facebookissa käytettyyn Flux-arkkitehtuuriin, jossa yhdensuuntainen datan kulku on toteutettu tapahtumien, lähettäjän, säiliön ja näkymän avulla. Kuvasta 12 nähdään, kuinka data kulkee edellä mainittujen komponenttien välillä.



Kuva 12. Eisenmanin (17, s. 294) esittämä kuvio datan kulusta Flux-arkkitehtuurissa.

Refluxin avulla yksinkertaistetaan Flux-arkkitehtuuria poistamalla lähettäjä. Näin ollen säiliö kuuntelee suoraan näkymästä käynnistettyjä tapahtumia. Näkymä puolestaan kuuntelee säiliössä tapahtuneita muutoksia ja näyttää päivittyneet tiedot käyttöliittymässä. Kuva 13 esittää insinöörityössä käytetyn Reflux-arkkitehtuurin datan kulkua.



Kuva 13. Eisenmanin (17, s. 294) esittämä kuvio datan kulusta Reflux-arkkitehtuurissa.

Stats Collector -sovelluksen data kerätään yksinkertaisten painikkeiden avulla. Esimerkiksi kotijoukkueen syöttöjen lukumäärän keräämiselle on oma painike, jonka sisällä näytetään syöttöjen lukumäärä. Napin painallus käynnistää tapahtuman, joka luo uuden syöttöobjektin säiliössä ja lisää sen kotijoukkueen syötöt sisältävään taulukkoon. Näkymä kuuntelee säiliötä ja näyttää siellä olevan syöttötaulukon koon. Esimerkkikoodista 1 nähdään, kuinka Stats Collector -sovelluksessa toteutettiin Reflux-arkkitehtuurin mukainen datan kulku kotijoukkueen syöttöjen lukumäärän keräämisessä.

```
// GameActions.js-tiedosto, jossa on listattu tapahtumat
export var gameActions = Reflux.createActions([
  .
  .
  "homePass",
  .
  .
]);

// GameStore.js-tiedosto, jossa määritellään säiliö
import { gameActions } from './../Actions/GameActions';

export default class GameStore extends Reflux.Store {
  constructor(props) {
    super(props);
    this.listenables = gameActions;
    this.state = {
      .
      .
      homePasses: [],
      .
      .
    };
  }

  onHomePass() {
    this.state.homePasses.push(new TimeEvent(
      this.state.minutes
    ));
    this.setState({homePasses: this.state.homePasses});
  }
}

// Game.js-tiedosto, jossa määritellään näkymä
import { gameActions } from './../Actions/GameActions';
import GameStore from './../Stores/GameStore';

export default class Game extends Reflux.Component {

  constructor(props) {
    super(props);
    this.store = GameStore;
  }

  render() {
    return (
      <TouchableHighlight onPress={gameActions.homePass}>
        <View>
          <Text>Pass</Text>
          <Text>{this.state.homePasses.length}</Text>
        </View>
      </TouchableHighlight>
    );
  }
}
```

Esimerkkikoodi 1. Reflux-arkkitehtuurin käyttäminen Stats Collector -sovelluksessa.

5.3 Komponentit

React Native -sovellukset koostuvat komponenteista. Komponentit sisältävät tarvittavat tiedot, joiden perusteella ne renderöidään näytölle. Yksinkertainen komponentti saattaa sisältää vain render-metodin, joka palauttaa JSX-syntaksia. Usein komponentit koostuvat myös toisista komponenteista. Sovelluksen rakenteen pilkkominen pieniin komponentteihin helpottaa sovelluksen ylläpitoa ja testausta.

Komponentit sisältävät kahdenlaisia muuttujia. Ominaisuus-muuttujat ovat komponentin luonnin yhteydessä välitettäviä parametreja. Tila-muuttujat ovat komponentin sisäisiä muuttujia, joiden arvo vaihtuu sovelluksen ajon aikana. Esimerkkikoodissa 2 on MainView-komponentti, joka koostuu erilaisista komponenteista kuten View, Image ja Text. React Nativen silta muuttaa komponentit kohdealustan vastaaviksi moduuleiksi. Esimerkiksi View-komponentti muutetaan iOS-alustalla UIViewksi.

```
export default class MainView extends Component {
  render() {
    return (

        <Image source={require('./../Images/background.png')}
        style={styles.mainBackground}>
          <View style={styles.logo}>
            <ResponsiveImage source={require('')} initWidth="" initHeight="" />
          </View>
          <View style={styles.graph}>
            <ResponsiveImage source={require('')} initWidth="" initHeight="" />
          </View>
          <View style={styles.mainBtnHolder}>
            <View style={styles.mainBtnContainer}>
              <TouchableOpacity style={styles.mainBtn} onPress={Actions.newGame}>
                <Text style={styles.mainBtnFont}>New Game</Text>
              </TouchableOpacity>
              <TouchableOpacity style={styles.mainBtn} onPress={Actions.savedGames}>
                <Text style={styles.mainBtnFont}>Statistics</Text>
              </TouchableOpacity>
            </View>
          </View>
        </Image>

    );
  }
}
```

Esimerkkikoodi 2. Stats Collector -sovelluksen MainView-komponentti.

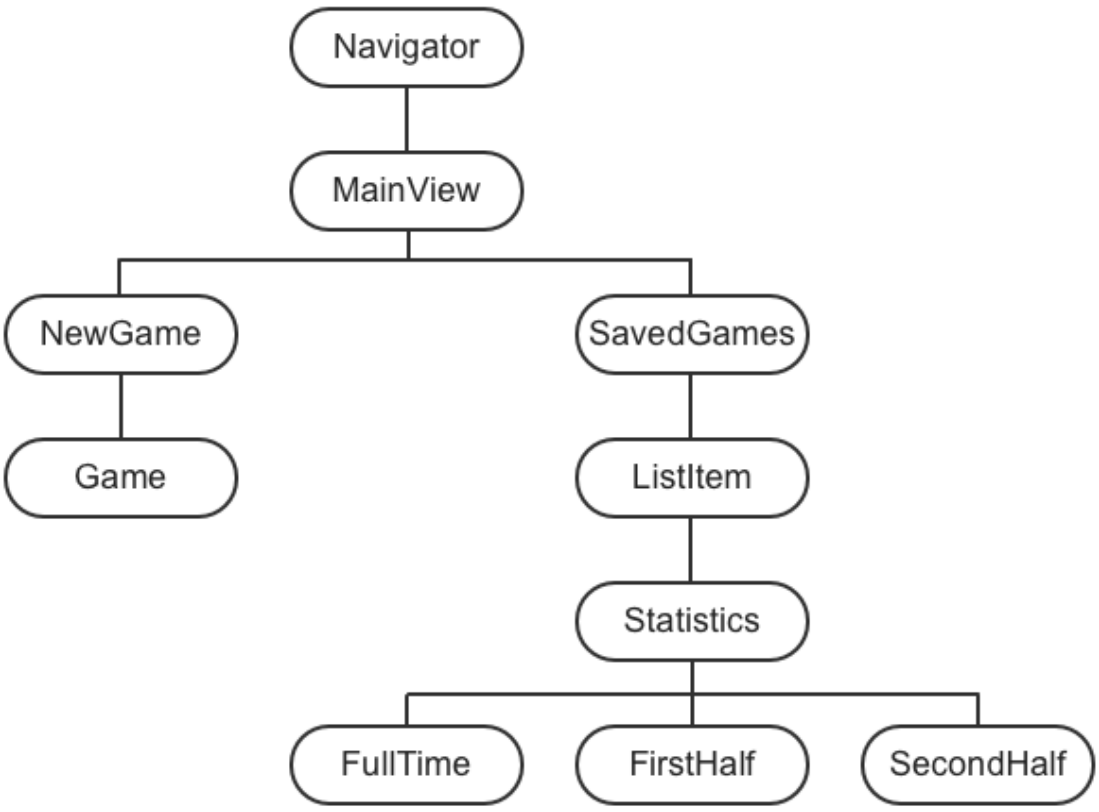
Esimerkkikoodista 2 nähdään myös, kuinka ResponsiveImage-komponentille välitetään parametrina initWidth- ja initHeight-ominaisuusmuuttujat. Komponentti voi hyödyntää ominaisuusmuuttujia toiminnassaan, kuten esimerkkikoodissa 3 on tehty. Siinä ListItem-komponentti asettaa pic-tilamuuttujan saamansa pic-ominaisuusmuuttujan perusteella.

Esimerkissä näkyy myös, kuinka pic-tilamuuttuja alustetaan komponentin constructor-metodissa.

```
export default class ListItem extends Component {

  constructor(props) {
    super(props);
    this.state = {
      pic: null
    };
  }

  componentDidMount() {
    this.setListImage();
  }

  setListImage() {
    if (this.props.pic === "Friendly") {
      this.setState({pic: require('./../Images/listFriendly.png')});
    } else if (this.props.pic === "Tournament") {
      this.setState({pic: require('./../Images/listTournament.png')});
    } else {
      this.setState({pic: require('./../Images/listLeague.png')});
    }
  }

  render() {
    return (
      .
      .
      .
    );
  }
}
```

Esimerkkikoodi 3. Stats Collector -sovelluksen ListItem-komponentti.

Kuten työssä aiemmin mainittiin, sovelluksen rakenne on tärkeä hyvän suorituskyvyn saavuttamiseksi. Stats Collectorin yksinkertaisuuden vuoksi rakenteella ei ole niin suurta vaikutusta suorituskykyyn. Jatkokehityksen kannalta jotkin komponentit voisi pilkkoa vielä pienempiin osakokonaisuuksiin. Kuva 14 kuvastaa Stats Collector -sovelluksen tämänhetkistä rakennetta. Kyseessä on korkean tason rakennekuvaus, sillä kaikki komponentit koostuvat useammasta komponentista. Liitteen 2 kuvista nähdään Stats Collector -sovelluksen rakenne loppukäyttäjän silmin.



Kuva 14. Stats Collector -sovelluksen korkean tason rakennekuvaus.

Sovelluksen näkymien luominen komponenttien avulla on tuttu lähestymistapa natiivikehittäjille. Android Studiolla näkymien luomiseen käytetään suoraan Androidin natiivikomponentteja. Näkymien määrittelyn voi toteuttaa XML-kielellä tai visuaalisella työkalulla, jossa komponentit raahataan hiirellä haluttuun paikkaan. Vaihtoehtoisesti näkymiä voi luoda myös Java-kielellä. Xcoden lähestymistapa iOS-laitteille on samankaltainen. Näkymiä voi luoda visuaalisella Storyboard-työkalulla tai ohjelmoida Objective-C- tai Swift-kielellä.

5.4 Tyylit

React Nativen tyylien kirjoittaminen muistuttaa hyvin paljon CSS:n syntaksia. Stats Collector -sovelluksen kehityksessä tyylien syntaksin tarkistaminen oli yksi eniten aikaa vieneistä osuuksista. Kuvasta 15 nähdään React Nativen syntaksin ja CSS:n samankaltaisuudet. Kehitystyö nopeutui huomattavasti ensimmäisten syntaksin tarkistamisten jälkeen, koska Nuclide IDE osasi ehdottaa aiemmin käytettyjä muuttujia.

React Nativen tyylien syntaksi

```
mainBtn: {
  justifyContent: 'center',
  alignItems: 'center',
  backgroundColor: 'rgba(255, 255, 255 ,0.3)',
  height: 70,
  width: 120,
  marginLeft: 10,
  marginRight: 10,
  borderRadius: 50,
  borderRightWidth: 3,
  borderColor: 'rgba(0, 0, 0, 0.5)',
},
```

CSS tyylien syntaksi

```
mainBtn {
  justify-content: center;
  align-items: center;
  background-color: rgba(255, 255, 255, 0.3);
  height: 70;
  width: 120;
  margin-left: 10;
  margin-right: 10;
  border-radius: 50;
  border-right-width: 3;
  border-color: rgba(0, 0, 0, 0.5);
},
```

Kuva 15. React Nativen ja CSS:n tyylien syntaksit.

Tyylejä on mahdollista käyttää usealla eri tavalla. Tyylit voi sijoittaa koodin joukkoon, JavaScript-tiedoston sisäisesti tai omaan tyylitiedostoon. Koodin joukkoon sijoitetut tyylit näyttävät sekavalta, jos tyylimääreitä on useita. Lisäksi koodin luettavuus kärsii ja samoja tyylejä ei voi hyödyntää vastaavissa komponenteissa. Esimerkkikoodi 4 havainnollistaa koodin joukkoon sijoitettuja tyylejä.

```
<View style={{flex: 1, justifyContent: 'center', alignItems: 'center'}}>
</View>

<View style={{flex: 1, justifyContent: 'center', alignItems: 'center'}}>
</View>
```

Esimerkkikoodi 4. Tyylien määrittely koodin joukossa.

Esimerkkikoodin 4 tapausta voidaan parantaa hyödyntämällä yhtä tyyliobjektia molemmissa View-komponenteissa. Tyyliobjektin voi määrittää JavaScript-tiedostossa, jolloin koodin luettavuus selkeytyy ja mahdollisia muutoksia ei tarvitse tehdä kahteen kertaan. JavaScript-tiedoston sisäiset tyyliobjektit toimivat pienissä komponenteissa, jotka eivät sisällä runsaasti tyylimuuttujia. Laajemmissa komponenteissa tyylit tuovat runsaasti lisää rivejä JavaScript-tiedostoon, jolloin kokonaisuutta on vaikeampi hahmottaa. Esimerkkikoodi 5 kuvastaa JavaScript-tiedoston sisäistä tyyliobjektia.

```
<View style={styles.container}>
</View>

<View style={styles.container}>
</View>

var styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: ’center’,
    alignItems: ’center’
  }
});
```

Esimerkkikoodi 5. Tyylien määrittely JavaScript-tiedostossa.

Stats Collector -sovelluksessa pyrittiin pitämään JavaScript-tiedostot mahdollisimman yksinkertaisina, minkä vuoksi tyyliobjektit sijoitettiin omiin tiedostoihin. Tyylitiedostot otettiin käyttöön JavaScript-tiedostoissa, jonka jälkeen tyyliobjekteja pystyttiin hyödyntämään. Kyseinen toteutustapa selkeyttää koodia ja helpottaa tyylien ylläpitoa. Stats Collector -sovelluksessa kullakin komponentilla on oma tyylitiedosto mutta laajemmassa sovelluksessa voisi hyödyntää myös yhteiskäyttöisiä tyylitiedostoja. Esimerkkikoodi 6 esittää Stats Collector -sovelluksessa käytettyä toteutusta tyylien määrittelyyn.

```
// JavaScript-tiedosto
import styles from './styles';

<View style={styles.container}>
</View>

<View style={styles.container}>
</View>


// Tyylitiedosto
import {StyleSheet} from 'react-native';

export default styles = StyleSheet.create({
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center'
  }
});
```

Esimerkkikoodi 6. Tyylien määrittely erillisessä tiedostossa.

5.5 Skaalautuvuus

React Nativella näkymien skaalautuvuus on toteutettu flexboxin avulla. Kyseessä on tapa, jolla määritellään komponenttien asettelu näytöllä. Sen avulla näkymät skaalautuvat erikokoisille laitteille ilman web-kehityksestä tuttuja media queryja. Flexbox on käytössä myös web-kehityksessä, mutta natiivikehittäjille sen käyttö voi olla vierasta.

Flexbox ei toimi täysin samalla tavalla CSS:n ja React Nativen välillä. React Nativella flex-ominaisuus saa arvokseen kokonaisluvun eikä tekstiä. Positiivinen kokonaisluku antaa komponentille suhteellisen määrän tilaa verrattuna muihin komponentteihin. Mikäli flex-ominaisuuden arvo on 0, komponentin koko määräytyy sen width- ja height-arvojen perusteella. Flex-ominaisuuden arvolla -1 komponentin koko määräytyy edellisen kohdan tavoin, mutta tarvitun tilan puuttuessa koko määräytyy minWidth- ja minHeight-arvojen mukaan. Kuva 16 havainnollistaa flex-ominaisuuden toimintaa positiivisilla arvoilla.

Kuva 16. Flex-arvon vaikutus komponentin saamaan tilaan.

Tavanomaisesti halutun lopputuloksen komponenttien asettelussa saavuttaa flexDirection-, alignItems- ja justifyContent-ominaisuuksien avulla. FlexDirection-ominaisuus

määrittää, näytetäänkö komponentit pysty- vai vaakasuunnassa. AlignItems- ja justifyContent-ominaisuudet määrittelevät komponenttien sijainnin pysty- ja vaakasuunnassa. Stats Collector -sovelluksessa käytettiin vain edellä mainittuja ominaisuuksia komponenttien asettelussa.

Eroavaisuuksia React Nativen ja CSS:n flexboxin välillä on myös ominaisuuksien oletusarvoissa. FlexDirection-ominaisuuden oletusarvona on pystysuuntainen asettelu, kun web-kehityksessä oletuksena on vaakasuuntainen asettelu. Lisäksi alignItems-ominaisuuden oletuksena komponentit venytetään sopimaan näytölle vaakasuunnassa, kun web-kehityksessä elementit asetetaan näytön yläreunaan.

Natiivikehityksessä näkymien skaalautuvuus vie paljon aikaa, sillä näkymät täytyy tehdä uudelleen erikokoisille laitteille. Android-kehityksessä näkymät luodaan näytön DPI:n perusteella, jolloin näkymiä täytyy luoda kuusi kappaletta. Androidin kuusi näytön tarkkuusluokkaa ovat

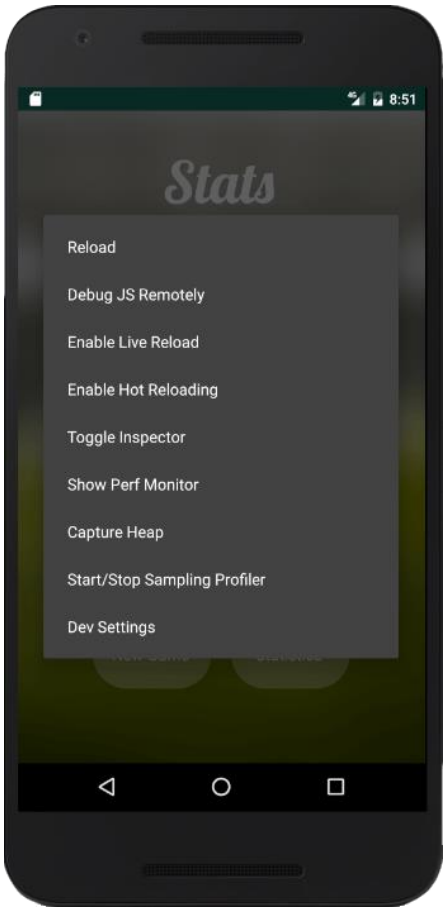
- ldpi (low)
- mdpi (medium)
- hdpi (high)
- xhdpi (extra-high)
- xxhdpi (extra-extra-high)
- xxxhdpi (extra-extra-extar-high).

Aiemmin kaikkien kuvien täytyi tukea kuutta eri näyttötarkkuutta. Vektorigrafiikan käyttö poistaa tämän ongelman. iOS-kehityksessä on samankaltaisia ongelmia, minkä vuoksi React Nativen käyttäminen nopeuttaa huomattavasti sovelluskehitystä.

5.6 Työkalut

React Nativeen sisältyy hyödyllisiä työkaluja, joiden avulla kehitystyö on nopeampaa ja virheiden löytäminen helpompaa. Kuvassa 17 nähdään Android-emulaattorilla käytössä olevat työkalut. Stats Collectorin kehityksessä hyödynnettiin jatkuvasti Chromen kehitystyökaluja, Live- ja Hot Reloadia sekä React Nativen sisäistä elementtien tarkistustyöka-

lua. Natiivikehittäjien ei ole pakko opetella uusia työkaluja, vaan he voivat ajaa sovelluksen Android Studiolla tai Xcodella ja käyttää niiden debuggaus-työkaluja. Web-kehittäjät voivat puolestaan käyttää tuttua Chromen kehitystyökalua.

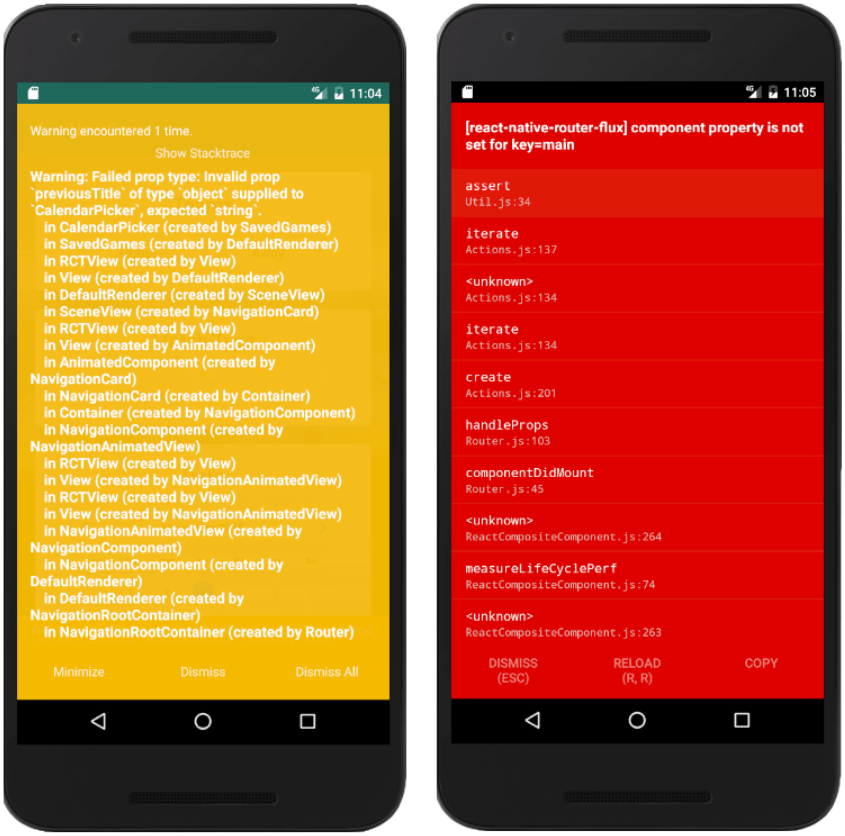


Kuva 17. React Nativen työkaluvalikko Android-emulaattorissa.

Mikäli kehittäjä haluaa nopeuttaa kehitystyötään, voi hän automatisoida sovelluksen päivittämisen ottamalla käyttöön Live Reload -työkalun. Sen ansiosta sovellus päivittyy joka kerta automaattisesti, kun muuttunut kooditiedosto tallennetaan. Koodimuutosten näkeminen laitteessa vaatii vain pari sekuntia kestävän sovelluksen päivityksen.

Hot Reloading toimii lähes samalla tavalla kuin Live Reload. Kun muuttunut kooditiedosto tallennetaan, sovellus hakee viimeisimmät muutokset JavaScript-kokoelmasta mutta ei uudelleenkäynnistä koko sovellusta. Hot Reloadingin ansiosta sovelluksessa säilyy senhetkinen tila. Tämä on hyödyllinen ominaisuus etenkin monimutkaisemmissa sovelluksissa, joissa komponenteilla on runsaasti eri tiloja tai ominaisuuksia.

Selkeät virhe- ja varoitusilmoitukset ovat tärkeitä ongelman löytämiseksi. Niiden esittäminen on toteutettu React Nativessa onnistuneesti. Mikäli koodi sisältää virheen, emulaattorin tai puhelimen näytöllä näytetään virheen aiheuttanut ongelma punaisella taustalla. Mikäli sovellus pystyy toimimaan ongelmasta huolimatta, näytöllä näytetään varoitus keltaisella taustalla. Insinöörityön sovelluskehityksessä virheet olivat yksinkertaisia ja niihin löytyi joka kerta ratkaisu virheilmoituksen kuvauksesta. Kehitysvaiheessa varoitukset voi laittaa pois näkyvistä mutta niiden käyttö on suositeltavaa. Sovelluksen tuotantoversiossa virheilmoituksia ei näytetä ollenkaan. Kuvassa 18 nähdään esimerkki React Nativen virheilmoitusnäkymistä.



Kuva 18. React Nativen varoitus- ja virhenäkymät Android-emulaattorissa.

Koska hyvä suorituskyky on tärkeä osa React Nativen toimintaa, sen seuraaminen on tehty vaivattomaksi. Perf Monitor-työkalun avulla kehittäjä näkee jatkuvasti sovelluksen FPS:n sekä pääsäikeessä että JavaScript-säikeessä. Suorituskyvyn seurannan ansiosta kehittäjä voi havaita suorituskykyyn eniten vaikuttavat toimenpiteet ja optimoida sovelluksensa suorituskykyä paremmaksi. Kuvasta 19 nähdään, kuinka Stats Collectorin FPS on lähes koko ajan 60.

Kuva 19. React Nativen Perf Monitor -työkalu Android-emulaattorin oikeassa yläreunassa.

React Native sisältää myös Inspector-työkalun, jonka avulla voidaan tutkia elementtejä. Vastaavanlainen työkalu on käytettävissä web-kehityksessä mutta natiivikehityksessä käytetyt ohjelmointiympäristöt eivät sellaista sisällä. Inspector on hyödyllinen erityisesti sovelluksen elementtien sijoittelun sekä tyylien asettelun apuna. Kuvassa 20 nähdään Stats Collector -sovelluksen päänäkymä Inspector-työkalulla tutkittuna.

Kuva 20. Stats Collector -sovelluksen päänäkymä Inspector-työkalulla tutkittuna.

6 Yhteenveto

Insinöörityössä tutustuttiin mobiilisovelluskehityksen eri toteutustapoihin ja verrattiin niiden hyviä ja huonoja puolia. Uusien entistä suorituskykyisempien ohjelmistokehysten ansiosta hybridisovellusten suosio on kasvanut. Suorituskykyyn vaikuttaa erityisesti natiivikomponenttien käyttäminen WebView-säiliön sijaan. Ohjelmistokehykset kuten Xamarin ja React Native hyödyntävät puhelimen natiivikomponentteja.

Työssä perehdyttiin erityisesti React Nativen toimintaan. Koska React Native pohjautuu Reactiin, oli siihen perehtyminen luontevaa aloittaa React JavaScript -kirjastosta. Uusina asioina opittiin muun muassa Reactin renderöintilogiikan toimintaa sekä sen taustalla toimivan algoritmin toimintaperiaatteita. React Native poikkeaa kuitenkin Reactista. Käytössä ei ole dokumenttioliomallia, vaan renderöinti piirtää natiivikomponentteja sovelluksen pääsäikeessä. JavaScriptin eristäminen pääsäikeestä edellyttää asynkronisia kutsuja, mikä tekee toteutuksesta haastavan.

Työn aiheeksi valittiin sovelluskehitys React Nativella, koska haluttiin tutustua uuteen ja nopeasti kehittyvään ohjelmistokehykseen. Tarkoituksena oli myös kehittää hybridisovellus, jolla on täysin yhteinen koodikanta Android- ja iOS-alustoille. Sovelluskehityksessä opittiin runsaasti React Nativen käytännöistä. Ympäristön asentaminen sujui vaivattomasti, mutta kehitystyön opettelu vaati aikaa ja kärsivällisyyttä. Erityisesti tyylien syntaksi vaati opettelua, koska se muistutti runsaasti CSS-syntaksia. Oppimista helpotti React Nativen jatkuvasti parantuva dokumentointi sekä yhteisön tuki. Kehitystyö React Nativella oli todella mielekästä ja sitä on helppo suositella.

Insinöörityön tuotoksena kehitettiin Stats Collector -mobiilisovellus, jolla voi kerätä tilastoja jalkapallo-ottelusta. Tarkoituksena oli kehittää sovellus, jolla on yhtenäinen koodikanta Android- ja iOS-alustoille. Kehitystyö oli huomattavasti nopeampaa verrattuna samanlaisen sovelluksen kehittämiseen erikseen molemmille alustoille. Lisäksi sovelluksen ylläpito on helpompaa yhden koodikannan ansiosta. Työlle asetetut tavoitteet toteutuivat ja sovellusta päästiin kokeilemaan sekä Android- että iOS-laitteilla.

Lähteet

1 Number of available apps in the Apple App Store 2008-2016. 2016. Verkkodokumentti. <https://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/>. June 2016. Luettu 18.10.2016.

2 Google Play: number of available apps 2009-2016. 2016. Verkkodokumentti. <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>. September 2016. Luettu 18.10.2016.

3 Schwaber, Ken & Sutherland, Jeff. 2016. The Scrum Guide. Verkkodokumentti. <http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf#zoom=100>. July 2016. Luettu 18.10.2016.

4 Vuorinen, Carl. 2014. Kolme tapaa kehittää mobiilisovellus. Verkkodokumentti. <http://w3.fi/kolme-tapaa-kehittaa-mobiilisovellus/>. Lokakuu 21, 2014. Luettu 19.10.2016.

5 Haapahovi, Sebastian. 2014. Mobiiliaplikaatioiden kehitys: HTML5, natiivi vai hybridi?. Verkkodokumentti. <http://www.klopal.fi/2014/09/01/mobiiliaplikaatioiden-kehitys-html5-natiivi-vai-hybridi/>. 1.9.2014. Luettu 20.10.2016.

6 Egham. 2016. Gartner Says Worldwide Smartphone Sales Grew 3.9 Percent in First Quarter of 2016. UK. Verkkodokumentti. <http://www.gartner.com/newsroom/id/3323017>. May 19, 2016. Luettu 19.10.2016.

7 Savuoja, Tuure. 2015. Hybridisovellukset mobiilisovelluskehityksessä. Espoo. Verkkodokumentti. <https://aaltodoc.aalto.fi/bitstream/handle/123456789/16648/master_Savuoja_Tuure_2015.pdf?sequence=1>. 28. huhtikuuta 2015. Luettu 20.10.2016.

8 Riippi, Juha. 2013. Natiivi, hybridi ja HTML5. Verkkodokumentti. <https://www.vincit.fi/blog/natiivi-hybridi-ja-html5/>. 27.5.2013. Luettu 20.10.2016.

9 Cheung, Harry. 2015. Mobile App Performance Redux. Verkkodokumentti. <https://medium.com/@harrycheung/mobile-app-performance-redux-e512be94f976#.zmcljczec>. Mar 17, 2015. Luettu 20.10.2016.

10 Rusila, Santeri. 2016. Alustariippumattomien mobiilisovellusten kehitystavat. Verkkodokumentti. <https://jyx.jyu.fi/dspace/bitstream/handle/123456789/50144/URN%3ANBN%3Afi%3Ajyu-201606062944.pdf?sequence=1>. 19. huhtikuuta 2016. Luettu 21.10.2016.

11 Friedman, Nat. 2016. Xamarin for Everyone. Verkkodokumentti. <https://blog.xamarin.com/xamarin-for-all/>. March 31, 2016. Luettu 21.10.2016.

12 Friedman, Nat. 2015. Xamarin Passes 1 Million Developer Milestone. Verkkodokumentti. <https://blog.xamarin.com/xamarin-passes-1-million-developer-milestone/>. April 29, 2015. Luettu 21.10.2016.

13 Knighton, Craig. 2016. Cross platform mobile development: When is Xamarin the best approach?. Verkkodokumentti. <http://mentormate.com/blog/xamarin-best-cross-platform-mobile-development-approach/>. March 30, 2016. Luettu 21.10.2016.

14 Occhino, Tom. 2015. React Native: Bringing modern web techniques to mobile. Verkkodokumentti. <https://code.facebook.com/posts/1014532261909640/react-native-bringing-modern-web-techniques-to-mobile/>. 26. maaliskuuta 2015. Luettu 21.10.2016.

15 Witte, Daniel & von Weitershausen , Philipp. 2015. React Native for Android: How we built the first cross-platform React Native app. Verkkodokumentti. <https://code.facebook.com/posts/1189117404435352/react-native-for-android-how-we-built-the-first-cross-platform-react-native-app/>. 14. syyskuuta 2015. Luettu 21.10.2016.

16 Laborde, Gant. 2016. Five Reasons Your Team Should Use React Native. Verkkodokumentti. <https://shift.infinite.red/five-reasons-your-team-should-use-react-native-d79b36b17e0b#.rywglfk9c>. Jan 22. Luettu 21.10.2016.

17 Eisenman, Bonnie. 2015. Learning React Native. O'Reilly Media.

18 The state of the Octoverse 2016. 2016. Verkkodokumentti. <https://octoverse.github.com/>. Luettu 21.10.2016.

19 Who's using React Native?. 2016. Verkkodokumentti. <https://facebook.github.io/react-native/showcase.html>. Luettu 22.10.2016.

20 Veikkaus katsoi React Native-kortin. 2016. Verkkodokumentti. <https://rnd.works/blog/veikkaus-react-native>. 11.3.2016. Luettu 22.10.2016.

21 Mihalcea, Romeo. 2014. How React.js works. Verkkodokumentti. <https://www.devcasts.io/p/how-reactjs-works/>. 29-SEP-2014. Luettu 26.10.2016.

22 Chedeau, Christopher. 2013. React's diff algorithm. Verkkodokumentti. <http://calendar.perfplanet.com/2013/diff/>. 28th Dec 2013. Luettu 25.10.2016.

23 Pandey, Harshit. 2015. ReactJS | Learning Virtual DOM and React Diff Algorithm. Verkkodokumentti. <http://www.oyecode.com/2015/09/reactjs-learning-virtual-dom-and-react.html>. September 21, 2015. Luettu 25.10.2016

24 Dispatch Queues. 2012. Verkkodokumentti. <https://developer.apple.com/library/content/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html>. Updated: 2012-12-13. Luettu 26.10.2016.

25 De Baets, Pieter; Lang Alexey; Sagnes, Frédéric; Tadeu, Zagallo. 2016. Dive into React Native performance. Verkkodokumentti. <https://code.facebook.com/posts/895897210527114/dive-into-react-native-performance/>. 28. maaliskuuta. Luettu 26.10.2016.

26 Tadeu, Zagallo. 2015. Bridging in React Native. Verkkodokumentti. <http://tadeuzagallo.com/blog/react-native-bridge/>. 14 Oct 2015. Luettu 26.10.2016.

27 Li, Sing. 2015. React Native: Into a new world of rapid iOS development. Verkkodokumentti. <http://www.ibm.com/developerworks/library/mo-bluemix-react-native-ios8/>. 15 July 2015. Luettu 26.10.2016.

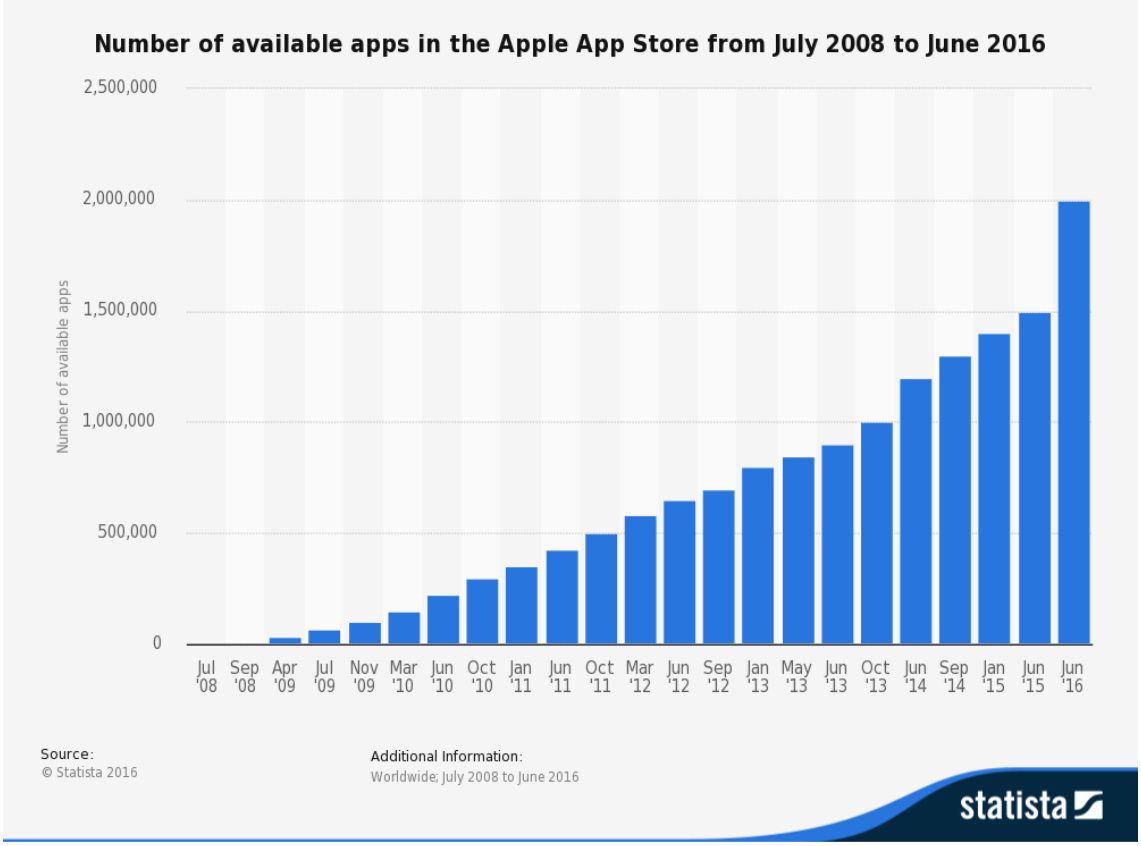
28 JavaScript Environment. 2016. Verkkodokumentti. <https://facebook.github.io/react-native/docs/javascript-environment.html>. Luettu 26.10.2016.

29 Vesza, József. 2016. JavaScriptCore Tutorial for iOS: Getting Started. Verkkodokumentti. <https://www.raywenderlich.com/124075/javascriptcore-tutorial>. April 7, 2016. Luettu 26.10.2016.

30 Getting Started. 2016. Verkkodokumentti. <https://facebook.github.io/react-native/docs/getting-started.html>. Luettu 2.11.2016.

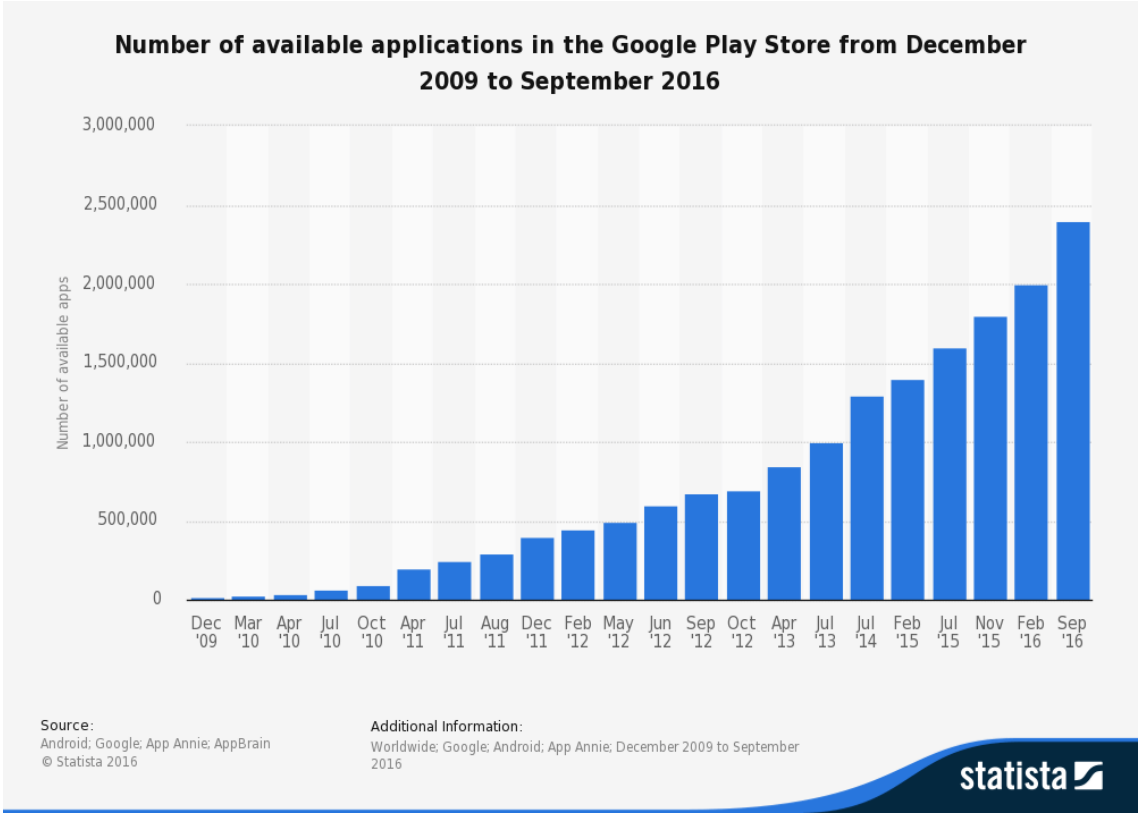
App Store- ja Google Play-sovelluskaupat

Applen perustama App Store on sovelluskauppa, jonka kautta iOS-laitteille asennetaan sovelluksia. Applen mobiililaitteisiin ei voi rehellisin keinoin asentaa sovelluksia muualta. Sovelluksen julkaiseminen App Storessa edellyttää kehittäjälisenssin ostamista, sekä sovellusten tarkistusprosessin läpäisyä. Kuvasta 1 nähdään App Storen sovellusten lukumäärän kehittyminen.



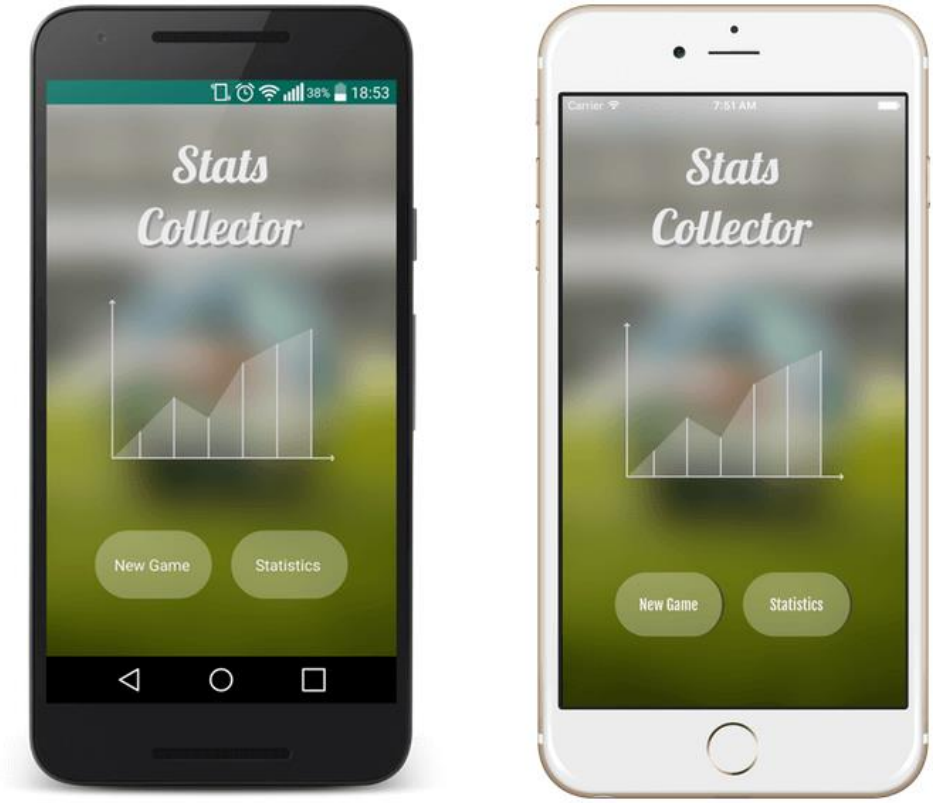
Kuva 1. Applen App Storen sovellusten lukumäärän muutos vuodesta 2008 (1).

Google Play on Googlen virallinen sovelluskauppa Android-laitteille. Laitteille voi asentaa sovelluksia myös muista lähteistä, kuten verkkosivut tai kolmannen osapuolen sovelluskaupat. Sovelluksen julkaiseminen edellyttää Google Play-kehittäjälisenssin ostamista. Google Playn sovellusten lukumäärä on kasvanut runsaasti viimeisen kuuden vuoden aikana (kuva 2).

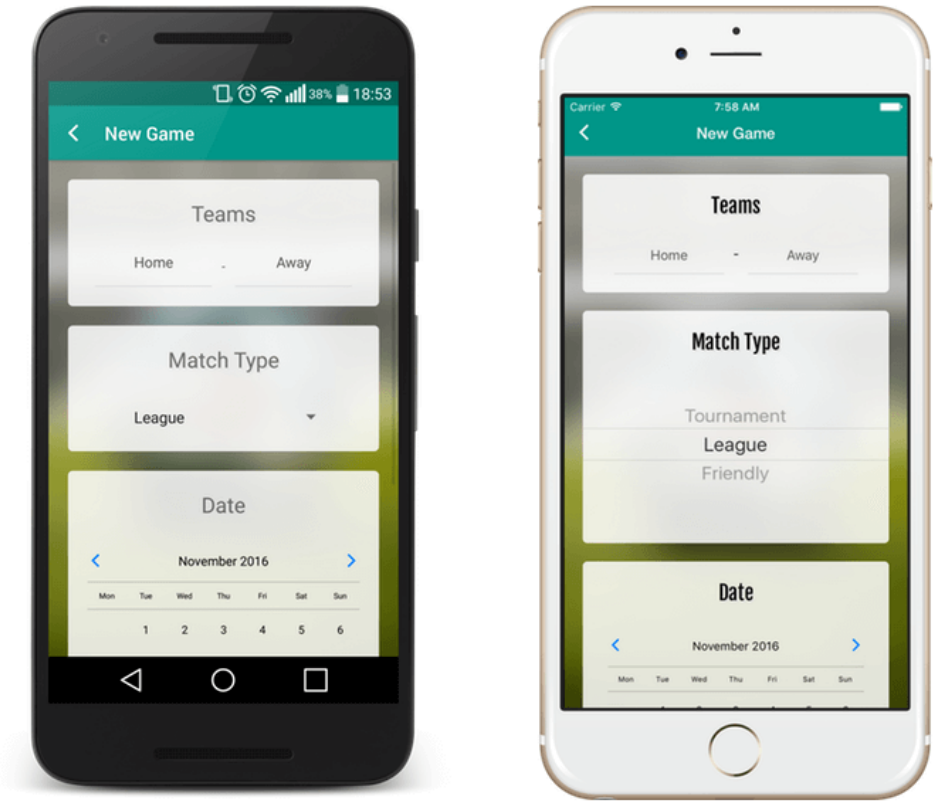


Kuva 2. Googlen Play-kauppapaikan sovellusten lukumäärän muutos vuodesta 2009 (2).

Stats Collector -sovelluksen näkymät

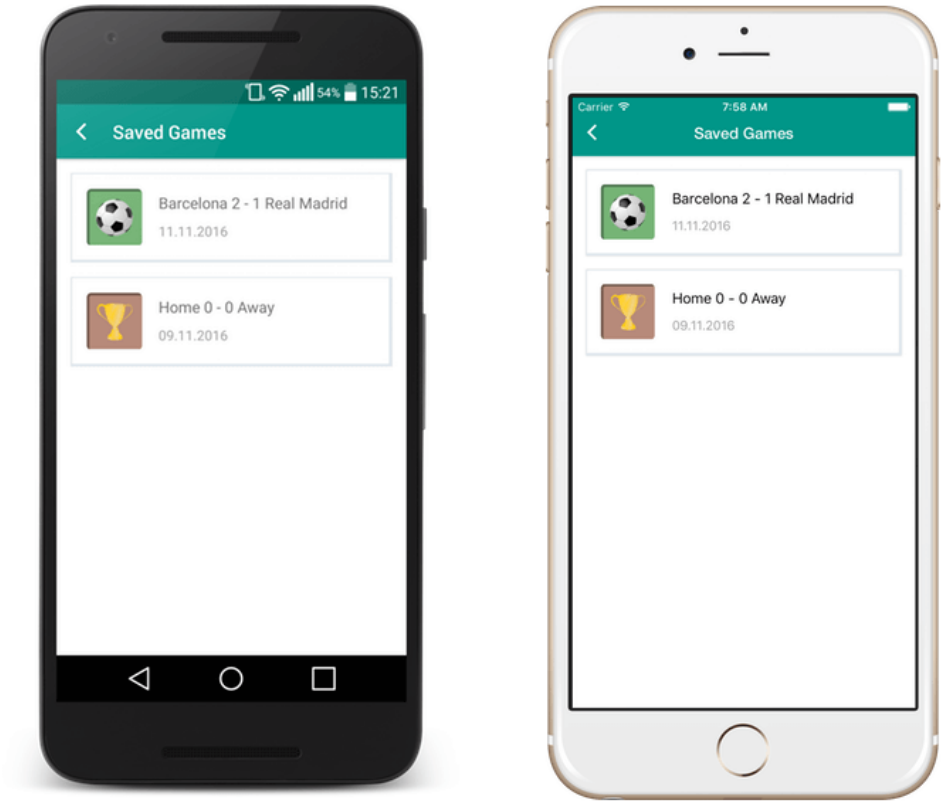


Kuva 1. Stats Collector -sovelluksen MainView-näkymä. MainView-näkymästä voi navigoida uuteen otteluun tai tarkastella aiemmin kerättyjä tilastoja.

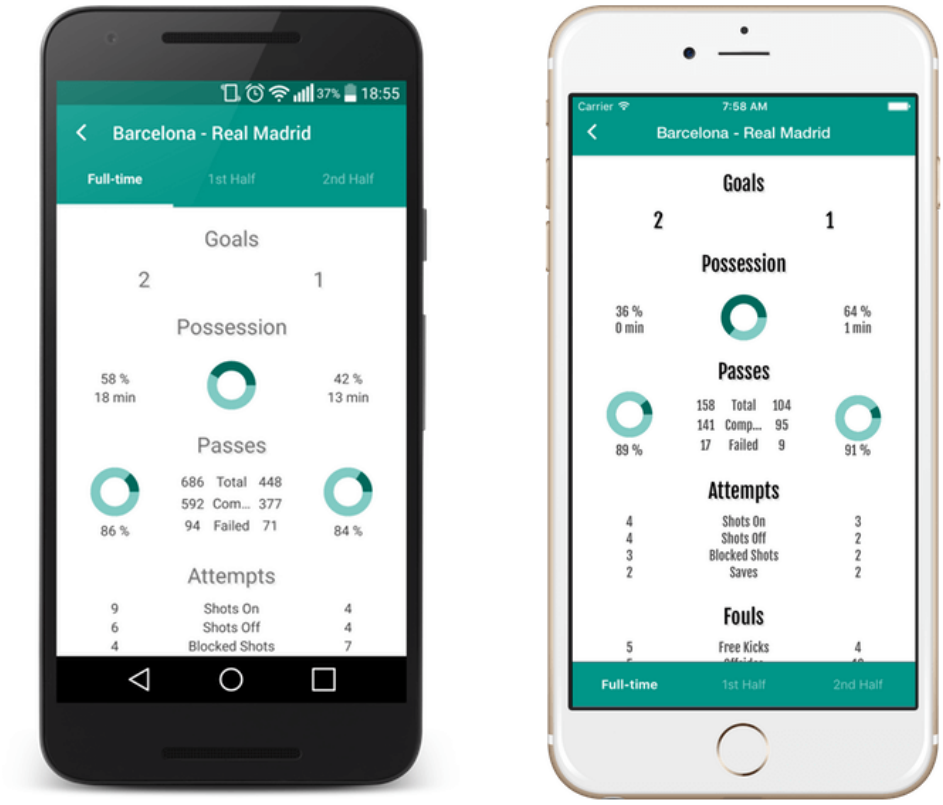


Kuva 2. Stats Collector -sovelluksen NewGame-näkymä. Tässä näkymässä syötetään tilastoitavan ottelun tiedot.

Kuva 3. Game-näkymän modaalinen ikkuna, jossa syötetään tiedot syntyneestä maalista. Game-näkymässä kerätään tilastoja ottelutapahtumia kuvaavien painikkeiden avulla. Aluekohtaisista tapahtumista kerätään lisäksi paikkatieto modaalisen ikkunan avulla.



Kuva 4. SavedGames-näkymä, joka koostuu ListItem-komponenteista. Näkymässä on listattu tilastoidut ottelut päivämäärän perusteella. ListItem-komponentin ikoni näyttää onko kyseessä ollut sarja-, turnaus- vai ystävyysottelu.



Kuva 5. Statistics-näkymä, joka koostuu FullTime-, FirstHalf- ja SecondHalf-komponenteista. Tämän näkymän kautta käyttäjä voi tarkastella ottelun tilastoja.