jamk.fi

React Native -animaatiot

Animaatiomenetelmien vertailu

Valtteri Puonti

Opinnäytetyö
Tammikuu 2020
Tekniikan ala
Insinööri (amk), tieto- ja viestintätekniikan koulutusala

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Puonti, Valtteri	Opinnäytetyö, AMK	Helmikuu, 2020
	Sivumäärä 40	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
React Native -animaatiot
Animaatiomenetelmien vertailu

Tutkinto-ohjelma
Insinööri (amk), tieto- ja viestintätekniikan tutkinto-ohjelma

Työn ohjaaja(t)
Ari Rantala. Esa Salmikangas.

Toimeksiantaja(t)
–

Tiivistelmä

Nykypäivänä älypuhelimia on enemmän kuin koskaan. Vastatakseen sekä iOS- että Android-käyttäjien tarpeisiin sovelluskehittäjien piti kehittää sovellukset molemmille alustoille. Erillisen sovelluksen kehittäminen molemmille alustoille ei kuitenkaan ollut aina business-näkökulmasta katsottuna järkevää.

Ratkaisuksi kehitettiin hybridiviitekehykset, joiden avulla kehittäjät pystyivät kääntämään yhden lähdekoodin monelle alustalle. Samaan aikaan, erottuakseen massasta, sovelluksen piti tarjota nykyaikaisia käyttöliittymäkokemuksia, joihin kuuluvat vahvasti eleet sekä animaatiot. Monet hybridiviitekehykset eivät kuitenkaan kyenneet tarjoamaan tätä kokemusta.

Työssä kartoitettiin, mitä mahdollisuuksia React Native-kehittäjällä on modernien käyttöliittymäkomponenttien luomiseen.

Eri menetelmien erojen selvittämiseksi päätettiin kehittää bottomsheet-komponentti. Kyseinen komponentti on yleisesti käytetty ja sisältää sekä eleitä että animaatoita. Työssä käyvät ilmi eri animaatiomenetelmien erot, viat, ja vahvuudet.

Tulokseksi saatiin suositus siitä, miten React Native -kehittäjän kannatta kehittää elepohjaisia ja animoitavia käyttöliittymäkomponentteja.

Avainsanat (asiasanat)
react, native, reanimated, animated, animointi, vertailu

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Puonti, Valtteri	Bachelor's thesis	February 2020
		Language of publication: Finnish
	Number of pages 40	Permission for web publication: x

Title of publication

React Native animations

Comparing different ways of animating components in React Native

Degree programme

Bachelor of engineering, Information Technology

Supervisor(s)

Rantala, Ari. Salmikangas, Esa.

Assigned by

—

Abstract

Nowadays there are more smartphones in the world than ever before. In order to cater to both iOS and Android users' needs, developers needed to make separate apps for both platforms. That, however, wasn't always a financially smart move.

As a solution, hybrid frameworks were developed. Hybrid frameworks enabled developers to compile a single codebase to run on multiple target platforms. However, in order to stand out from the masses, the apps needed to provide a modern user experience that usually included gesture and animation-based components. Many hybrid-frameworks could not provide the experience.

React Native stood out from the hybrid frameworks, as it used native UI components, instead of being another web-view based implementation.

In the thesis, options for creating a modern gesture-, and animation driven user experience in React Native are mapped. In order to provide a good testing ground for different approaches, a bottom sheet component was developed using the different approaches.

The pros and cons of the approaches were brought to light, and ultimately a recommendation for how to create modern UI-components in React Native was found.

Keywords/tags (subjects)

react, native, reanimated, animated, animation, comparison

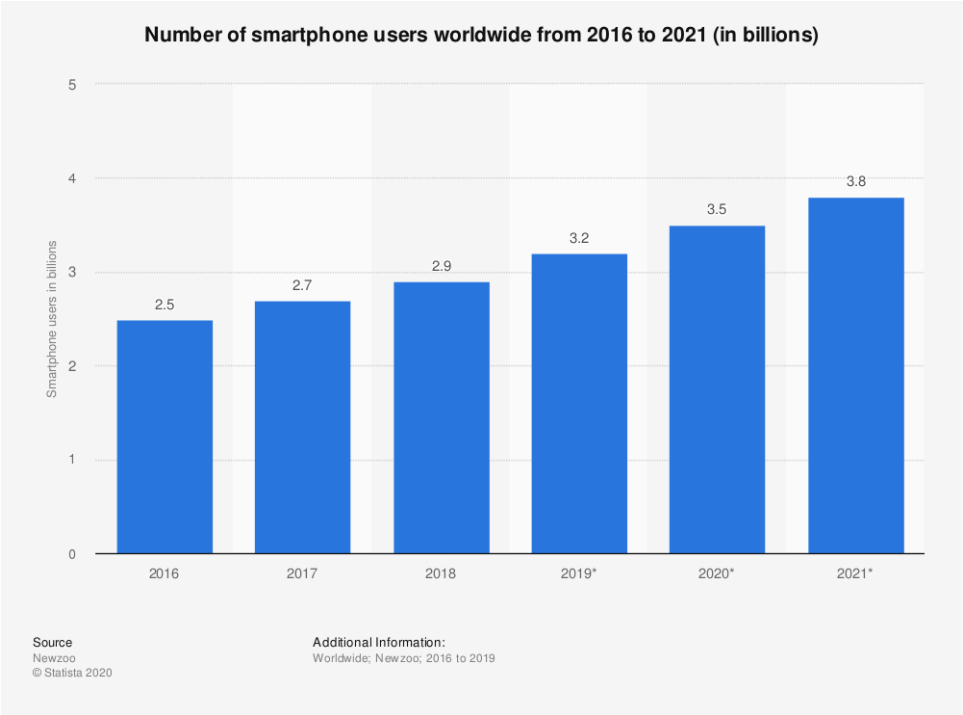
Miscellaneous (Confidential information)

Sisältö

Kuviot

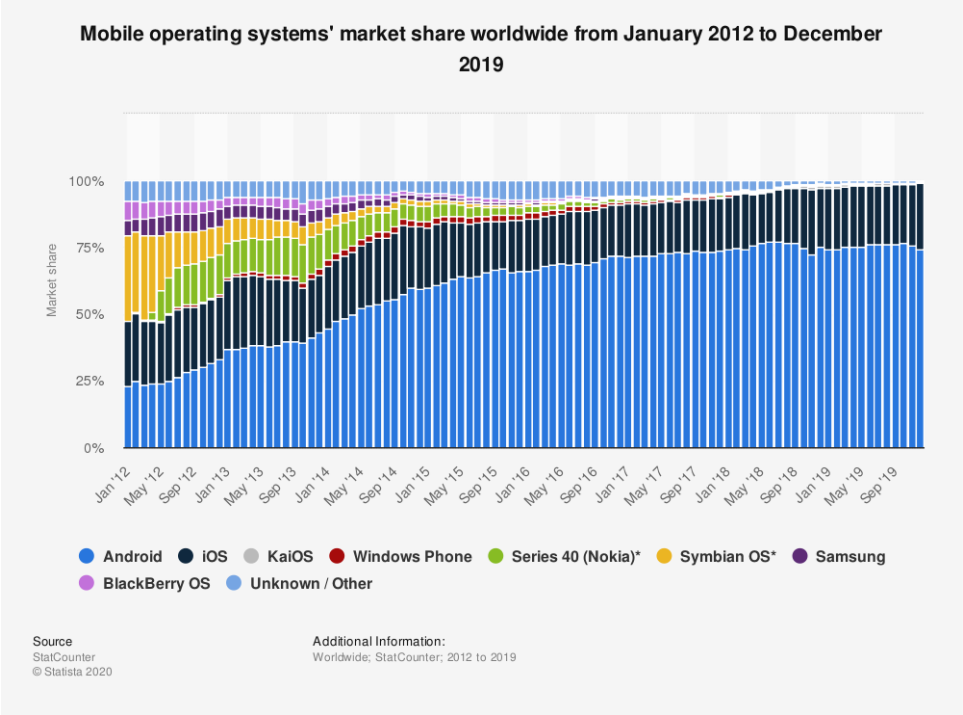
1 Johdanto

Newzoon (2019) ennusteen mukaan älypuhelinten käyttäjien odotetaan nousevan 3,8 miljardiin vuoteen 2021 mennessä (ks. Kuvio 1).



Kuvio 1. Älypuhelin käyttäjien maailmanlaajuinen lukumäärä miljardeina, vuodesta 2016 vuoteen 2021 (Newzoo 2019)

Määrä on valtava ja on kasvanut vuosi vuodelta. Tuosta määrästä lähes jokaisella on käytössään joko Android- tai iOS-pohjainen älypuhelin. (ks. Kuvio 2)



Kuvio 2. Mobiilikäyttöliittymien maailmanlaajuinen markkinaosuus tammikuusta 2012, joulukuuhun 2019 (StatCounter 2020)

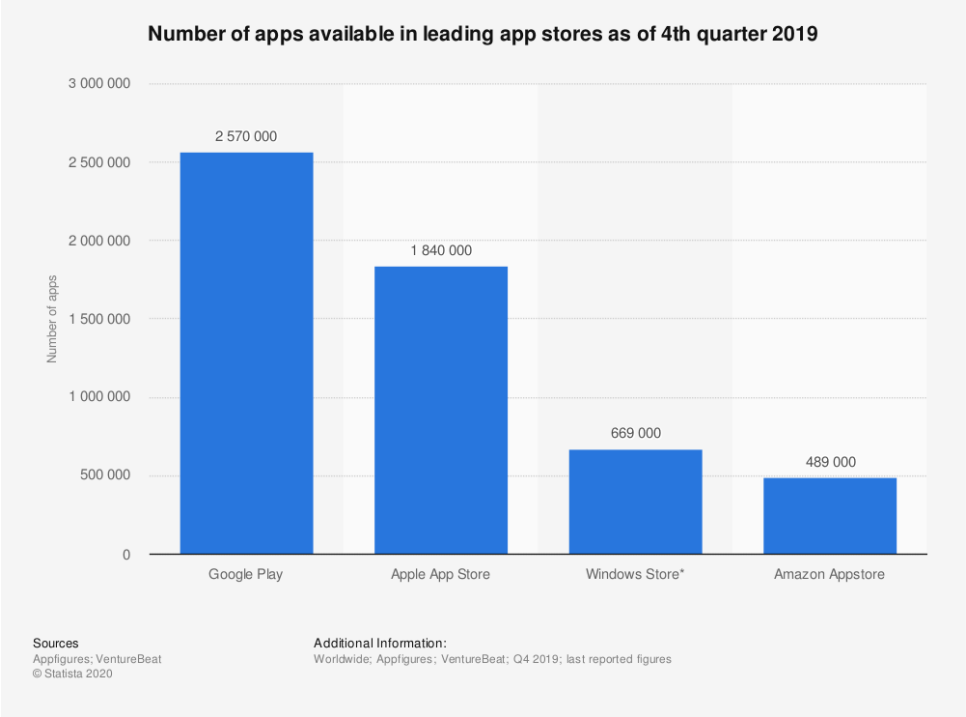
Aiemmin sovelluskehittäjien piti valita, kummalle alustalle he kehittävät sovelluksensa. Jos he halusivat tavoittaa sekä Android- että iOS-yleisön, heidän piti kehittää molemmille alustoille omat sovellukset. Koska sovelluskehitys on aikaa, rahaa ja osaamista vaativa prosessi, moni päättikin valita jommankumman alustan, mikä sulki pois ison siivun mahdollisesta käyttäjäkunnasta. Mobiililaitteelle skaalautuvat websivut toimivat tietyissä käyttötarkoituksissa hyvin, mutta ne ovat lähtökohtaisesti eri näköisiä ja tuntuisia kuin natiivi käyttöliittymäkomponenteilla rakennetut sovellukset.

Kyseiseen pulmaan kehitettiin ratkaisuksi hybridiviitekehykset, joita käyttämällä yhdellä kielellä kirjoitettu koodikanta kääntyy monella alustalla pyöriväksi sovellukseksi.

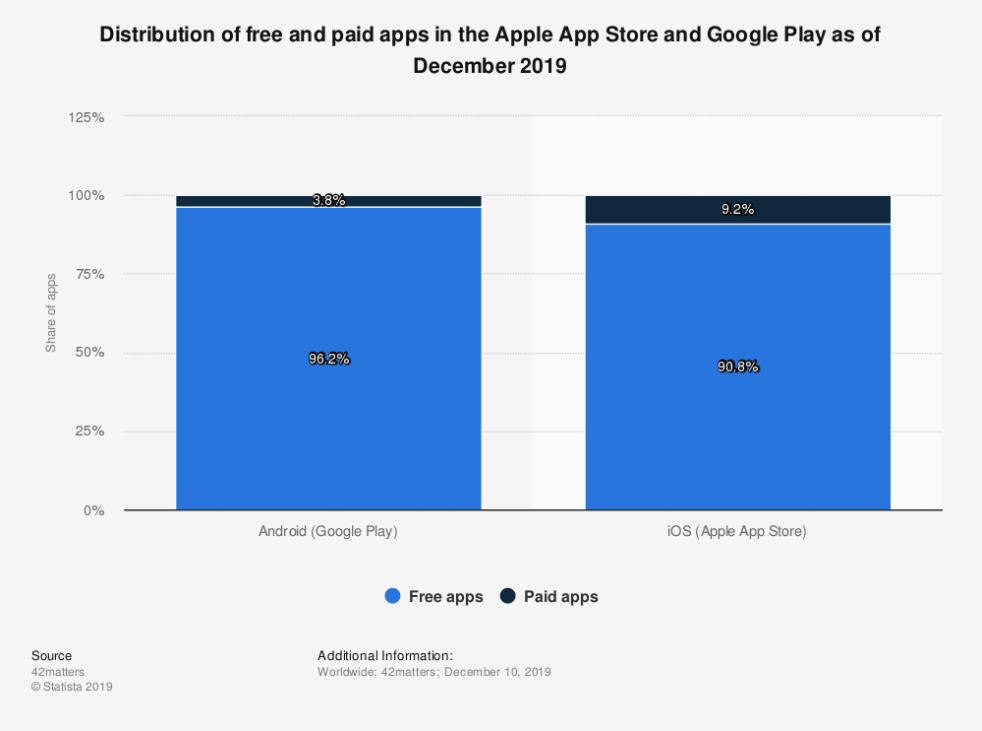
Vielä hetki sitten yleisin muoto hybridiviitekehyksistä oli web-pohjainen implementaatio, joissa HTML-komponenteista rakennettu Progressive Web Application pyöri joko Androidin WebView'ssä tai iOS:n UIWebView'ssä. Kommunikaatio natiivirajapintoihin tapahtui valitun viitekehyksen kautta, esimerkkinä Apachen Cordova.

Vaikka kyseiset viitekehykset tarjosivatkin mahdollisuuden kääntää web-sovelluksia “natiivi” sovelluksiksi, ne eivät olleet täydellisiä, ne eivät olleet oikeasti natiiveja. Yleisenä ongelmana olivat suorituskykyongelmat.

Suorituskyvyllä on väliä. Androidin sovelluskaupassa, Google Play Storessa, oli vuoden 2019 lopulla noin 2,5 miljoonaa sovellusta, joista ilmaisia oli 96.2 %. Applen App Storessa sen sijaan sovelluksia oli noin 1,8 miljoonaa, joista ilmaisia 90.8 %. (ks. Kuviot 3 ja 4).



Kuvio 3. Sovellusten määrä johtavissa mobiili sovelluskaupoissa vuoden 2019 lopussa (Appfigures & VentureBeat 2020)



Kuvio 4. Ilmaisten ja maksettujen sovellusten jakautuma Googlen Play Storessa ja Applen App Storessa joulukuussa 2019 (42matters 2019)

Kuluttajan kannalta on mahtavaa, kun on varaa valita. Valinnan määrä saa kuitenkin aikaan myös sen, että käyttäjän kynnys vaihtaa sovellusta on todella pieni. On elintärkeää, että sovellus toimii moitteitta ja on hyvännäköinen.

Yleisin tapa, jolla ilmaiset sovellukset tuottavat rahaa, on mainosten näyttäminen. Jotta mainoksia saadaan näytettyä paljon, käyttäjä pitää saada sitoutumaan sovellukseen. Pitää tarjota jotain, joka saa käyttäjän palaamaan sovelluksen pariin. Käyttökokemuksen ja hyvän käyttöliittymäsuunnittelun tärkeys korostuu.

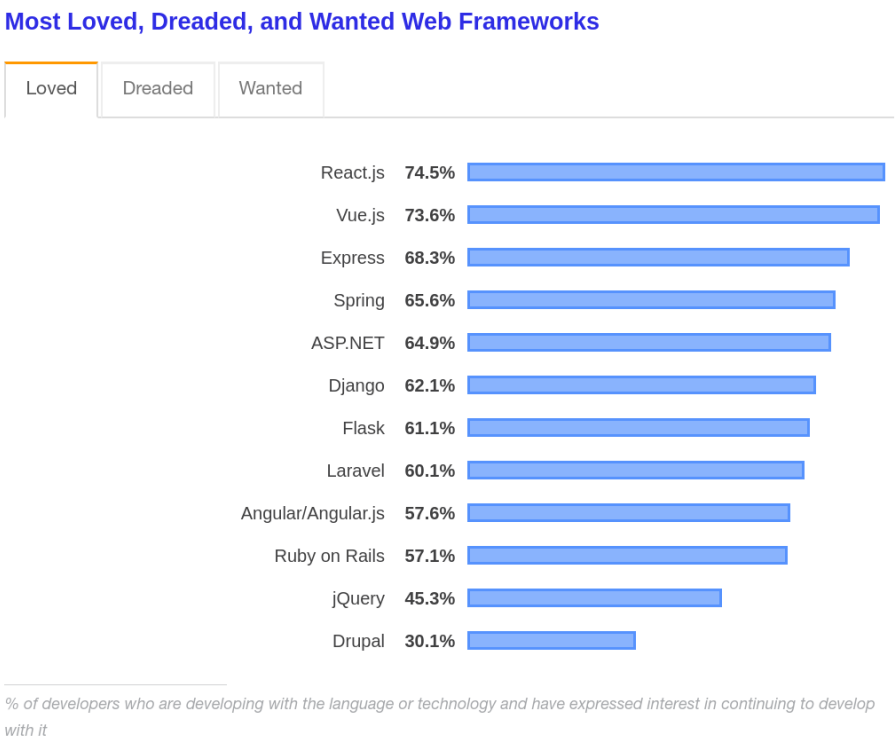
Työn tavoitteena oli löytää paras menetelmä käyttöliittymäanimaatioiden luomiseen React Native -teknologialla. Työ toteutettiin ohjelmoijan näkökulmasta, joten työn huomio kiinnittyy käytännön toteutusten implementoimiseen sekä toteutettujen animointimenetelmien vertailemiseen.

Työtä voidaan pitää tekemishetkellään ajankohtaisena katsauksena React Nativen mahdollisuuksiin näyttävien, animaatioita ja eleitä vaativien käyttöliittymien luomiseen. Työtä voidaan siis toivottavasti pitää eräänlaisena oppaana projektissaan React Nativen käyttöä harkitsevalle kehittäjälle.

2 React

2.1 Yleisesti

React on Facebookin kehittämä komponentteihin perustuva käyttöliittymäkirjasto. Reactia kirjoitetaan JavaScriptillä tai TypeScriptillä. React on ollut frontend-kehittäjien suosiossa julkaisustaan saakka, eikä hiipumisen merkkejä ole havaittavissa. React oli rakastetuimpien web-viitekehyksien listan kärjessä StackOverflown vuoden 2019 kehittäjäkyselyssä (ks. Kuvio 5). (Developer Survey Results 2019).

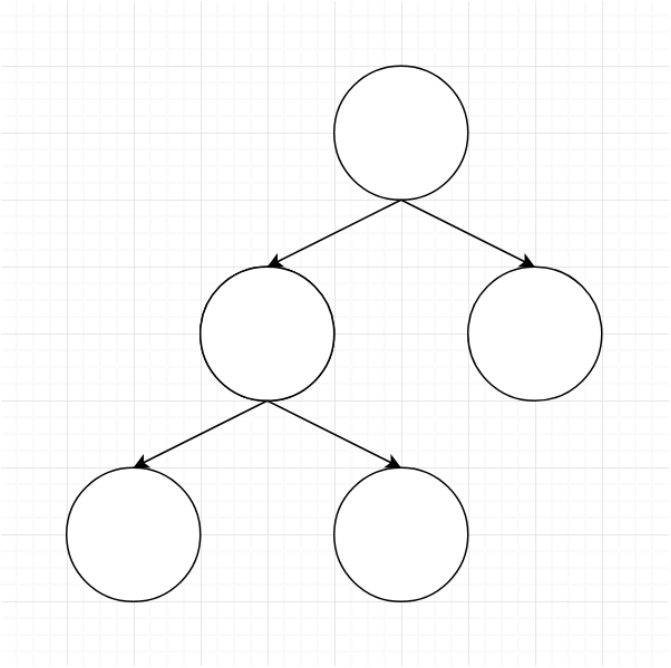


Kuvio 5. Prosentuaalinen määrä kehittäjistä, jotka ovat kehittäneet kyseisellä teknologialla, ja ilmaisevat haluaan jatkaa kehittämistä kyseisellä teknologialla (Developer Survey Results 2019)

React-sovellus koostuu komponenteista. Komponentti on Reactin perusrakennuspalikka. Komponentti on pohjimmiltaan kokoelma tyyliteltyjä käyttöliittymäelementtejä. Esimerkkinä komponentti nimeltään LabeledButton voisi koostua tekstikentästä ja napista ja ButtonGroup-komponentti ryhmästä LabeledButton-komponentteja.

React-komponenteista rakentuu puu, jonka juuressa on React-sovelluksen sisäänkäynti ja latvassa HTML-elementtejä. (ks. Kuvio 6)

Komponenteista rakentuva käyttöliittymä on hyvä siksi, että kerran ohjelmoitua komponenttia voi käyttää monessa paikassa uudestaan. Koodia tarvitsee kirjoittaa vähemmän, ja käyttöliittymä pysyy yhtenäisemmän näköisenä.



Kuvio 6. React sovelluksen pelkistetty rakenne

2.2 Tilattomat ja tilalliset komponentit

Reactissa voidaan määrittää tilallisia ja tilattomia komponentteja.

Tilaton komponentti on yksinkertaisimmillaan funktio, joka palauttaa kokoelman käyttöliittymäelementtejä. Aivan kuten normaalitkin funktiot React-komponentit voivat ottaa vastaan argumentteja, “propseja”, eli ominaisuuksia (ks. Kuviot 7 & 8).

Tilaton komponentti päivittyy vain, jos sille lähetettyjen propsien muutos aiheuttaa käyttöliittymämuutoksia.

```
function Hello(props) {
  return <p>Hello {props.name}</p>;
}
const Hola = (props) => <p>Hola {props.name}</p>;
```

Kuvio 7. Funktionaaliset (tilattomat) React-komponentit

```
class App extends React.Component {
  render() {
    return (
      <div className="App">
        <Hello name={"World"} />
        <Hola name={"Amigos"} />
      </div>
    );
  }
}
```

Kuvio 8. Propsien lähettäminen funktionaalisille komponenteille

Tilallisella komponentilla on nimensä mukaisesti tila. Tila on JavaScript-objekti, jonka kenttien päivittäminen setState-funktiolla johtaa komponentin uudelleen piirtämiseen. Tilalla on olennainen rooli Reactissa. Tilaa hyödyntämällä ohjelmoija saa hallinnan käyttöliittymäkomponenteista.

Yleisin esimerkki tilallisesta komponentista on laskuri, jossa napin painallus kasvattaa laskurin lukemaa (ks. Kuvio 9).

```
class Counter extends React.Component {
  state = {
    count: 0
  }
  render() {
    return(
      <div>
        <p>Current count: {this.state.count}</p>
        <button
          onClick={() =>
          this.setState({
            count: this.state.count+1
          })} >
            Add one
          </button>
      </div>
    )
  }
}
```

Kuvio 9. Tilallinen React-komponentti

Button-elementtiin lisätty onClick-funktio käyttää setState-funktiota laskurin lukeman päivittämiseen. Tila muuttuu, mikä johtaa komponentin uudelleen piirtämiseen.

2.3 Virtual DOM

DOM eli Document Object Model on rajapinta, joka mahdollistaa HTML- tai XML-dokumenttien käsittelemisen kokoelmana olioita. DOM mahdollistaa HTML- ja XML-elementtien manipuloimisen skriptauskielillä, kuten JavaScriptillä. (Introduction to the DOM 2020.)

Virtual DOM on Reactin käyttämä käsite DOMista, joka ei näy selaimen ruudulla, vaan pysyy käyttäjälle näkymättömissä tietokoneen muistissa. React käyttää Virtual DOMia käyttöliittymämuutosten tehokkaaseen päivittämiseen.

Jos React ei käyttäisi Virtual DOMia, jokainen React-sovelluksen juurikomponentin tilanpäivitys johtaisi koko sovelluksen uudelleenpiirtämiseen. Uudelleenpiirtäminen on raskas operaatio, ja niin kannattaa tehdä vain komponenteille, jotka sitä vaativat.

Reactin tilanpäivitykset lasketaan ensin Virtual DOMiin, minkä jälkeen virtuaalisen ja selaimessa näkyvän DOMin eroavaisuudet lasketaan tekniikalla nimeltä Reconciliation (Reconciliation 2019). Tekniikan avulla React päivittää vain ne noodit, jotka vaativat päivittämistä.

Koska React-komponentin tilanpäivitys ei vaikuta suoraan DOM-puuhun, tilanpäivitykset ovat luonnostaan asynkronisia. Tilanpäivitysten asynkronisuutta, ja sen haittoja animoimisessa, käsitellään luvussa 5.4.

3 React Native

React Native on Facebookin kehittämä hybridiviitekehys, jonka avulla Reactilla kirjoitettu koodi kääntyy sekä iOS- että Android-käyttöjärjestelmillä toimiviksi sovelluksiksi. Erona normaaliin Reactiin on se, että React Nativessa käytetään omia käyttöliittymäkomponentteja HTML-elementtien sijaan. Divin tilalla on View, erinäisten tekstielementtien tilalla on Text (ks. Kuvio 10).

Nämä React Nativen käyttöliittymäkomponentit kääntyvät kompilaatiovaiheessa sekä iOS:n että Androidin natiiveiksi käyttöliittymäkomponenteiksi.

Se, että React Native käyttää natiivi käyttöliittymäkomponentteja, on React Nativen vahvuus verrattuna muihin hybridiviitekehyksiin. Tarve imitoida natiivi käyttöliittymäkomponenttien ulkonäköä ja toiminnallisuutta on poistettu.

```
class HomeScreen extends Component {
  public render() {
    return (
      <View style={[styles.container]}>
        <Text>Home screen</Text>
      </View>
    )
  }
}
```

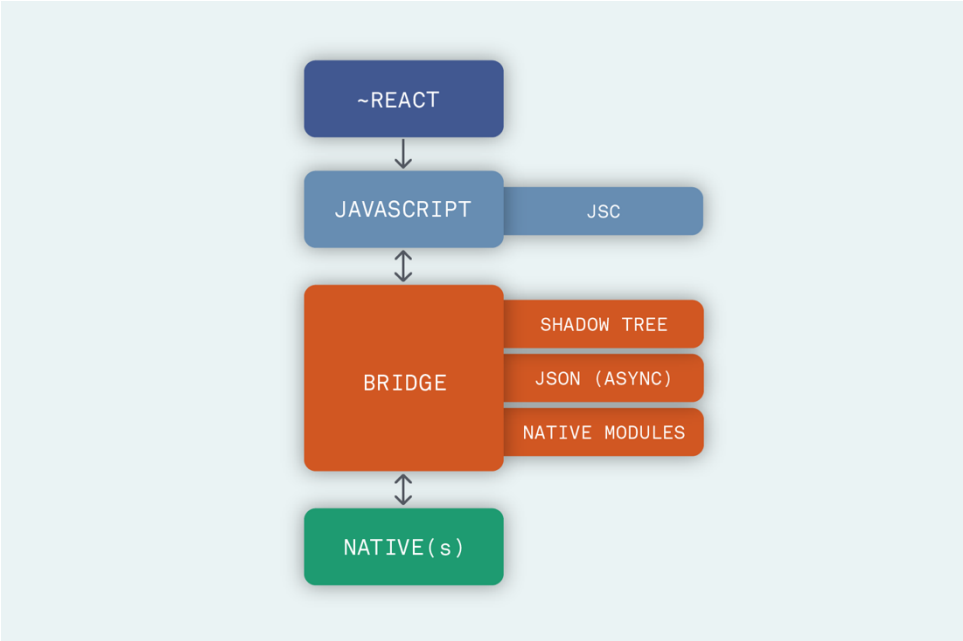
Kuvio 10. React Native komponentti

React Native toimii niin, että varsinainen React-sovellus pyörii omassa JavaScript-threadissä. React Native käyttää JavaScriptin ajamiseen Applen Safari-selaimen JavaScriptCore-moottoria. Androidista löytyvä V8-JavaScript moottori ei ole yhteensopiva React Nativen kanssa, minkä vuoksi React Native joutuu paketoimaan JavaScriptCore-moottorin sovelluksen mukaan Androidille käännettäessä kasvattaen sovelluksen kokoa. (Gaba & Ramachandran 2018, luku React Native Internals).

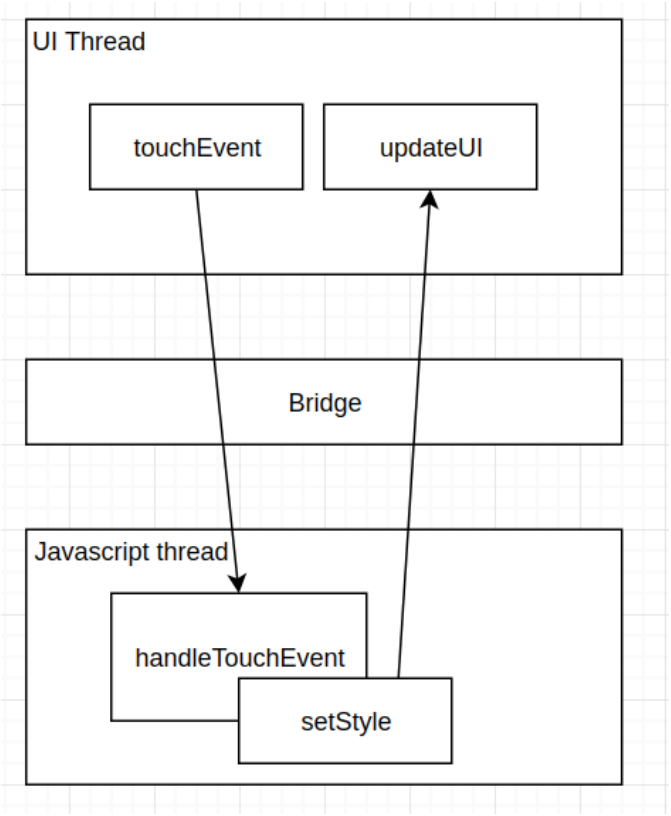
JavaScript ei pysty kommunikoimaan suoraan natiivipuolen kanssa, minkä vuoksi React Native joutuu käyttämään puolten yhdistämiseksi ”bridgeä” eli siltaa (ks. Kuvio 11). Silta on sama Androidilla sekä iOS:lla. Saman sillan käyttäminen molemmilla alustoilla onnistuu, koska silta on kirjoitettu C++:lla. C++ kääntyy konekieleksi, jota jokaisen puhelimen prosessori osaa suorittaa. JavaScript ja C++ eivät jaa samoja tietotyyppejä. Siksi sillan läpi kulkeva data pitää serialisoida ennalta määrättyyn muotoon siltaan lähetettäessä. (Sciandra 2019).

Natiivipuolen kutsuminen C++:sta onnistuu iOSlla ilman ongelmia, mutta Androidilla kutsujen pitää mennä Java Native Interfacen (JNI) kautta. JNI:n läpi meneminen tarkoittaa yhtä lisäkerrosta, jonka läpi data pitää serialisoida. Android NDK dokumentaation ”JNI vinkit”-sivulla mainitaan tärkeimmäksi suorituskykyvinkiksi JNI-metodien kutsumisen frekvenssin ja läpikulkevan datan serialisoinnin minimoimisen. (JNI Tips 2020.)

Datan kulku käyttöliittymästä takaisin JavaScriptiin kulkee samaa reittiä kuin käyttöliittymäpäivitykset, mutta toiseen suuntaan (ks. Kuvio 12).



Kuvio 11. React Nativen pelkistetty arkkitehtuuri (Sciandra 2019)



Kuvio 12. Datan kulku React Nativessa

4 Animointi

Animointi on pohjimmiltaan kuvien liikuttamista. Elokuvaprojektorit toimivat 24 Hz:n päivitysnopeudella, mikä tarkoittaa, että uusi kuva päivittyy vanhan tilalle 24 kertaa sekunnissa. Normaaleissa tietokoneissa ja puhelimissa näytön päivitysnopeus on 60 Hz.

60 Hz tarkoittaa yhtä ruudunpäivitystä per 16 millisekuntia. 16 millisekuntia on aika paljon aikaa tietokoneen prosessorin ajassa, mutta on hyvin mahdollista, että raskaampien operaatioiden välissä uusi ruutu ei ehdi päivittyä sille asetetussa aikarajassa, mikä johtaa siihen, että sovelluksen käyttöliittymä näyttää pätkivän. Edellä mainittuja raskaita operaatioita ovat I/O-operaatiot, verkkokutsut, raskaat laskutehtävät ja isojen tietorakenteiden käsittelyt.

Jokainen on varmasti kohdannut elämässään hitaan sovelluksen tai verkkosivun, jonka seurauksena hiiren liike näyttää pätkivän. Kyseinen pätkiminen johtuu siitä, että tietokoneen prosessori on niin kiireinen muiden tehtävien kanssa, ettei se ennätä pitää ruutua päivitettynä.

Ruudunpäivitystahdin ei tarvitse laskea paljon, jotta käyttäjä huomaa selvän eron. Jopa yksi kadonnut ruutu riittää huomattavaan nykäisyyn, siksi kehittäjien prioriteettina onkin suorituskyvyn optimoiminen. Käyttäjän ei pitäisi nähdä raskaista datankäsittelyoperaatioista muuta kuin 60 Hz:n tahdilla päivittyvä käyttöliittymä.

5 Animaatiomenetelmien erojen havainnollistaminen

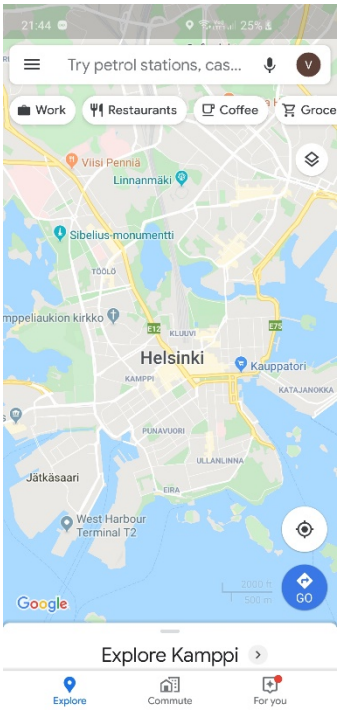
5.1 Testiasetelma

Eri animaatiomenetelmien erojen havainnoimiseksi kehitettiin Bottomsheet-komponentti käyttäen eri menetelmiä. Jokaisen implementaation keskeisenä palikkana oli ”react-native-gesture-handler” -kirjasto, joka tarjoaa hyvät puitteet käyttäjän kosketusten vastaanottamiseen ja käsittelyyn.

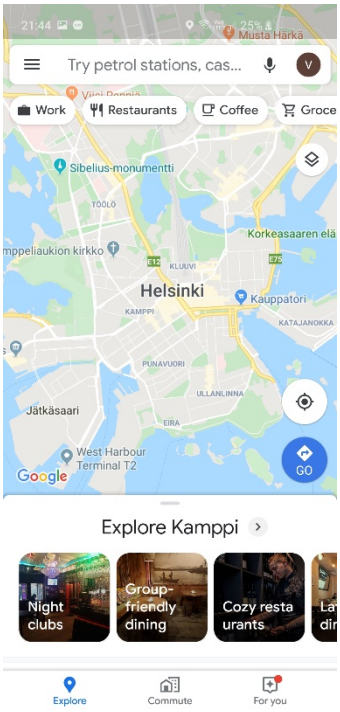
5.2 Bottomsheet-komponentti

Bottomsheetillä tarkoitetaan avattavaa ja suljettavaa käyttöliittymäkomponenttia, jonka tarkoituksena on tuoda lisää sisältöä ruudulle peittämättä koko ruutua. Hyvä esimerkki bottomsheet komponentista löytyy Google Mapsista.

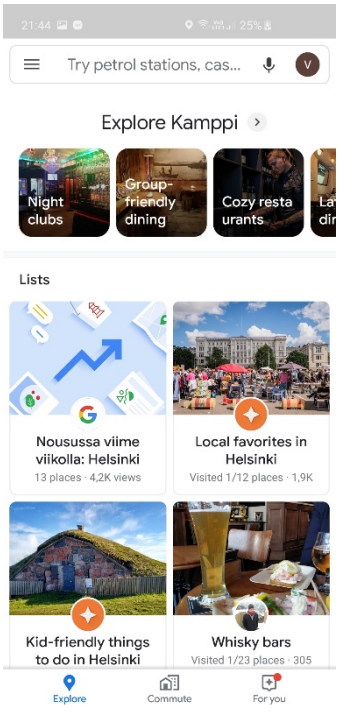
Google Mapsin bottomsheetillä on neljä tilaa: piilotettu, suljettu (ks. Kuvio 13), raotettu (Kuvio 14) ja aukinainen (Kuvio 15). Tilojen välinen navigointi tapahtuu komponentin otsikkoa klikkaamalla tai vetämällä. Irti päästettäessä komponentti animoituu lähimpään mahdolliseen tilaan.



Kuvio 13. Suljettu tila



Kuvio 14. Raotettu tila



Kuvio 15. Avattu tila

5.3 Bottomsheetin implementoiminen React Nativessa

Työssä bottomsheet-komponentti on oletuksena koko ruudun täyttävä näkymä, jota siirtämällä alla oleva näkymä saadaan näkyviin. Ruudun täyttämiseksi sheetin sijainti asetettiin absoluuttiseksi ja näkymän tyyliominaisuudet ***top***, ***bottom***, ***left*** ja ***right*** arvot asetettiin nollaksi, jolloin sheetin jokainen reuna on 0 pikselin päässä emonäkymän reunoista.

Sheetin alla olevan näkymän näyttämiseen löytyy monta keinoa.

Yksi tapa on ***top*** arvon muuttaminen, jolloin sheetin yläreuna siirtyy asetetun pikselimäärän päähän sheetin emonäkymän yläreunasta.

Toinen tapa on ***marginTop*** arvon muuttaminen, jolloin sheetin yläreunan ja emonäkymän yläreunan välinen rako kasvaa halutun määrän.

Kolmas tapa on transformaatio, joka siirtää koko näkymää haluttuun suuntaan halutun määrän.

Työssä käytettiin transformaatiota kaikissa paitsi yhdessä esimerkissä. Valinta perustuu siihen, että elementin sijaintiin ja kokoon vaikuttavat muutokset ovat raskaampia kuin pelkkä siirtäminen. Reaktiivisessa käyttöliittymässä, kuten React Nativen käyttämässä flexbox-implementaatiossa, elementin koon muutokset vaikuttavat sisar- ja lapsielementtien kokoon.

Sheetin aukaiseminen ja sulkeminen hoidetaan testattavissa implementaatioissa joko klikkaamalla tai klikkaamalla ja vetämällä riippuen valitun implementaation tuomista mahdollisuuksia.

Luvuissa 5.4-5.7 käydään läpi eri vaihtoehdot, jotka React Native -kehittäjällä on käytössään tyyliominaisuuksien animoimiseen.

5.4 Reactin oma tila

Reactin tilan päivittäminen on yleisin tapa päivittää käyttöliittymää React sovelluksissa. Kuviossa 9 näkyvässä esimerkissä <Counter/>-komponentin count-tilan päivittäminen johti siihen, että käyttöliittymään päivittyi isompi numero jokaisella napin painalluksella. Voitaisiinko tilaa käyttää käyttöliittymän animointiin? Mitä jos komponentin count-arvoa käytettäisiin sheetiä siirtävän tyyliominaisuuden arvona, ja tilan päivittämisen hoitaisi napin painalluksen sijaan jokin toistuva funktio?

Ideana hyvä, mutta käytännössä huono.

Jos muistellaan aiemman React-kappaleen Virtual DOM:ia käsittelevää alakappaletta, muistetaan, että tilanpäivitykset eivät päivity suoraan käyttöliittymään. Sen sijaan, muutoksista rakennetaan uusi versio muistiin, ja diffaus-algoritmin perusteella päätetään, mitkä komponentit päivitetään.

React Nativen tapauksessa datan pitää kulkea myös sillan läpi, joka lisää tilanpäivityksen ja näytön päivittymisen välistä aikaa.

Kyseisten operaatioiden ajaminen ennen jokaista ruudunpäivitystä ei ole mahdollista. Siksi työn käytännön osuuden ”tilan päivitys”-kohtaan toteutettiin vain yksinkertainen näkymä, jossa bottomsheetin headerin klikkaaminen vaihtaa komponentin isOpen-tilaa päälle ja pois. IsOpen tila määrittää bottomsheetin sijainnin ruudulla käyttäen yksinkertaista “jos auki, niin sijainti on X, jos kiinni, niin sijainti on Y”-lausetta.

Vaikka näkymässä ei olekaan varsinaista animaatiota, näkymä toimii hyvänä pohjana tulevia animaatiomenetelmiä varten.

5.5 LayoutAnimation API

React Native sisältää kokeellisen LayoutAnimation API:n tilan muutosten tuomien ulkoasumuutosten animoimiseen. Ohjelmoijan pitää vain määrittää yksinkertainen animaatio konfiguraatio, ja päivittää komponentin tila. (ks. Kuvio 16) React Native hoitaa loput. Tuloksena on natiivisti ajettu, nätti animaatio.

```
LayoutAnimation.spring()
this.setState({
  isOpen: !this.state.isOpen,
})
```

Kuvio 16. Animaation aloittaminen LayoutAnimation kirjastolla

LayoutAnimation API ei rasita siltaa eikä JavaScript puolta jatkuvilla päivityskäskyillä, koska itse animointi hoidetaan natiivipuolella. JavaScript puolella hoidetaan vain animaation konfigurointi, jonka jälkeen animaation ohjeet lähetetään natiivipuolelle yhdellä käskyllä.

LayoutAnimation API on mahtava tapa lisätä animaatioita käyttöliittymämuutoksiin, jotka vaikuttavat komponenttien kokoon ja sijaintiin. LayoutAnimation APIa ei voi käyttää transformaatioiden animoimiseen. Siksi työn käytännön esimerkissä transformaation sijaan bottomsheetin siirtämiseen käytetään ***marginTop*** ominaisuutta.

Esimerkkejä LayoutAnimation API:n "normaaleista" käyttötarkoituksista ovat esineen lisääminen listaan tai lyhennetyn näkymän liu'uttaminen kokomittaiseksi. Käytännössä siis tilanteita, joissa halutaan siirtyä pisteestä A pisteeseen B ilman välimuotoja.

Koska LayoutAnimation API on vielä työn tekemisen aikaan kokeellisessa tilassa, sovellukseen pitää lisätä yhden rivin mittainen konfiguraatio, jotta toiminnallisuus saadaan käyttöön Androidilla. (ks. Kuvio 17)

```
// tslint:disable-next-line: no-unused-expression
UIManager.setLayoutAnimationEnabledExperimental &&
  UIManager.setLayoutAnimationEnabledExperimental(true)
```

Kuvio 17. LayoutAnimationin käyttöönotto Androidilla

5.6 Animated

5.6.1 Yleisesti

React Nativen coreen kuuluu Animated-kirjasto, joka sisältää työkaluja käyttöliittymän animoimiseen.

Animated kirjaston perustana on Animated.Value. Animated.Value on arvo, jota voidaan animoida käyttämällä Animated kirjastosta löytyviä metodeja, kuten spring(), timing() ja decay(). Animated.Valueilla voidaan tehdä aritmeettisia operaatioita, ja niitä voidaan interpoloida luoden laajan alan käyttötarkoituksia.

Animated.Valuesta ei olisi paljoa hyötyä, jos sitä ei saisi käytettyä näkymien tyylien määrittelyyn. Siksi Animated-kirjastossa tuleekin mukana oma View-komponentin implementaatio, jolle voidaan tehdä niin.

Animated-kirjasto mahdollistaa bottomsheetin aukaisemisen ja sulkemisen vetämällä sekä painamalla. Jotta sheetin koko liikerata saadaan implementoitua, sheetiä pitää miettiä kahdessa eri tilassa: kun sitä vedetään ja kun siitä päästetään irti.

Kun sheetiä vedetään, se seuraa sormea. Sheetin sijainnin funktion voidaan silloin määritellä olevan lähtöpiste + vetoliikkeen kulkema matka.

Bottomsheet komponentin tilaan määritettiin muuttujat baseOffset ja panY, ja bottomsheet-komponentin root-view:n Y:n suuntaisen offsetin arvoksi asetettiin kyseisten arvojen summa. Summaus tehdään käyttämällä Animated.add()-funktiota (ks. Kuvio 18).

```
public render() {
  const { baseOffset, panY } = this.state
  const sheetOffset = Animated.add(baseOffset, panY)
  return (
    <Animated.View
      style={[
        styles.container,
        {
          transform: [
            {
              translateY: sheetOffset,
            },
          ],
        },
      ]}
    >
```

Kuvio 18. Animated bottomsheetin transformaation laskeminen

PanY:n arvo saadaan käyttämällä Animated kirjaston tarjoamaa event()-metodia. Kyseiseen arvoon eristetään PanGestureHandlerin onGestureEvent-callbackin palauttaman objektin translationY-ominaisuus, joka kuvastaa kuinka paljon sormi on liikkunut Y-akselin suuntaisesti vetoliikkeen aikana. (ks. Kuvio 19)

```
<PanGestureHandler
  onGestureEvent={event([
    {
      nativeEvent: {
        translationY: this.state.panY,
      },
    },
  ])}
```

Kuvio 19. Vetoliikkeen käsittely Animatedin event-metodilla

Tässä vaiheessa animaation ensimmäinen osa toimii. Sheet seuraa sormea vetäessä, mutta se ei animoidu lähimpään ankkuriin irti päästettäessä. Sheet jää siihen, mihin se jätettiin. Jos sheetiä koitetaan vetää uudestaan, huomataan että sheet hyppää alkuasentoon, josta se lähtee seuraamaan sormen liikettä. Tämä johtuu siitä, että panY lähtee uuden vetoliikkeen alkaessa nollasta. Jotta animaatiosta saadaan halutun kaltainen, PanGestureHandleriin pitää lisätä onHandlerStateChange-callback, jota kutsutaan, kun vetoliikkeen tila muuttuu.

Vetoliikkeen päätyttyä lasketaan sheetin sijainti vetoliikkeen päättyessä, sekä animoidaanko sheet joko avattuun vai suljettuun tilaan. Sijainnin laskemisen jälkeen varsinainen animoiminen on todella helppoa kiitos kuviossa 20 näkyvän spring()-metodin.

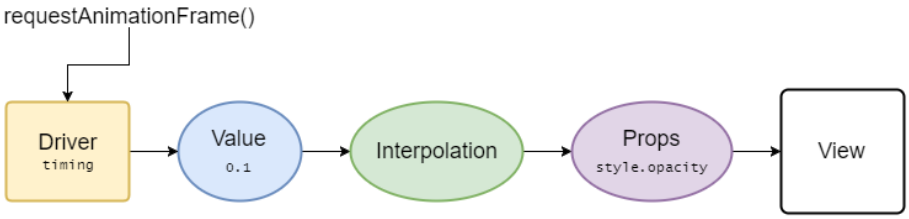
```
Animated.spring(this.state.baseOffset, {
  toValue: destination,
  velocity: velocityY,
}).start()
```

Kuvio 20. Animated kirjaston spring-metodi

Animoimiseen päätettiin käyttää spring-metodia, koska PanGestureHandlerin onHandlerStateChange palauttaa myös vetoliikkeen Y-akselin suuntaisen nopeuden irti päästettäessä, ja spring-funktion konfiguraatioon voidaan asettaa nopeus, jolla animaatio lähtee liikkeelle. Transitio vetoliikkeestä spring-animaatioon on luonnollisen näköinen, kun animaatio jatkuu samalla nopeudella kuin irti päästettäessä.

5.6.2 useNativeDriver

Animated kirjaston animaatioajuri käyttää sisäisesti Web-ohjelmoinnista tuttua requestAnimationFrame-metodia (ks. Kuvio 21). Metodin tarkoituksena on ilmoittaa ohjelmaa tulevasta ruudunpäivityksestä.



Kuvio 21. Animated kirjaston ajurin toimintaperiaate

JavaScript on single-threaded kieli, eli kaikki operaatiot tapahtuvat yhdellä threadillä. Täten, threadin blokkaavat operaatiot vaikuttavat koko sovelluksen toimintaan. On siis mahdollista, että samaan aikaan tapahtuvat raskaat tiedonkäsittelyoperaatiot johtavat siihen, että ruudunpäivityskäskyä ei keretä prosessoida riittävän nopeasti.

Ongelman ratkaisemiseksi React Native -kehittäjät kehittivät tavan lähettää koko animaatio data yhdellä kutsulla käyttöliittymäpuolelle. Tämä ominaisuus saadaan käyttöön asettamalla useNativeDriver-parametri “true”:ksi. Animated kirjaston event-metodi tukee myös useNativeDriver-parametria, joka tarkoittaa sitä, että PanGestureHandlerin vetoliikkeen arvojen siirtäminen Animated.Valueen voidaan hoitaa täysin käyttöliittymäpuolella.

Lisäämällä useNativeDriver-parametri PanGestureHandlerin käsittelevään Animated.event()-metodiin, sekä vetoliikkeen päättymisen jälkeiseen Animated.spring()-animaatioon, JavaScript-puolelle jää vain vetoliikkeen jälkeinen animaation määränpään löytäminen ja animaation aloittaminen.

5.7 Reanimated

5.7.1 Yleistä

React Native Reanimated on ulkopuolisen Software Mansion ohjelmistotalon ylläpitämä React Native kirjasto. Reanimatedilla kirjoitetut animaatiot pyörivät sata prosenttisesti käyttöliittymä-threadillä, joka johtaa huomattaviin suorituskyky parannuksiin. Se on kuin useNativeDriver-parametri olisi laitettu kaikkeen.

Reanimatedin Animated-nimiavaruus tarjoaa samankaltaisia luokkia kuin normaali Animated. Animated kirjaston peruspalikat, Animated.View ja Animated.Value löytyvät. Samoin Animated.event()-funktio.

Toisin kuin Animated kirjasto, Reanimatedin ideana ei ole tarjota mahdollisimman yksinkertaista tapaa ajaa animaatiota, Reanimatedin ideana on antaa kehittäjälle mahdollisimman matalan tason kontrolli animaatioista. Tarjotakseen tämän kontrollin, Reanimated antaa kehittäjälle laajan kattauksen Animated.Node:ja, jotka ovat yksinkertaisuudessaan aritmeettisia ja loogisia operaattoreita. Reanimatedin juju onkin siinä, että animaatiot kirjoitetaan matalalla tasolla deklaratiiviseen tyyliin.

Aivan kuten React Nativen View- ja Text-komponentit, Reanimatedin Animated.Node:t käännetään natiivi vastineiksi.

Animaatiologiikan siirtäminen kokonaan käyttöliittymäpuolelle on hyvä juttu, koska nyt JavaScript-puoli voi keskittyä täysin business-logiikan pyörittämiseen, ja käyttöliittymäpuoli voi keskittyä hienon, animoidun käyttöliittymän näyttämiseen.

5.7.2 Animaation valmistelu

Animaation valmistelu tapahtuu lähes samalla tavalla kuin kappaleen 5.6 Animated-esimerkissä.

Sheetin toiminnallisuuden kannalta oleellisimmat arvot siirretään Animated.Valueihin Reanimatedin event()-metodia käyttämällä.

Kyseiset arvot lähetetään panAnimation-funktiolle parametreina. Funktio palauttaa Animated.Node:n, joka vastaa yksin koko animaation kulusta. (ks. Kuvio 22)

```
private sheetOffset: Animated.Node<number> = panAnimation(
  this.position,
  this.panState,
  this.panY,
  this.velocityY,
  this.toggleTapAnim,
  this.anchor
)

public render() {
  return (
    <Animated.View
      style={[
        styles.container,
        {
          transform: [
            {
              translateY: this.sheetOffset,
            },
          ],
        },
      ]}
    >
```

Kuvio 22. Reanimated noodin alustaminen

5.7.3 Yleistä nodeista ja vetämisen implementoiminen

Kuten tässä kappaleessa aiemmin mainittiin Reanimated animaatio koostuu kokoelmasta noodeja. Yleisimpiä noodeja ovat ”block”, ”eq”, ”cond” ja ”set”.

”Block”-noodi on kokoelma noodeja. Kokoelman jokainen noodi käsitellään järjestyksessä, jonka jälkeen se palauttaa viimeisimmän noodin arvon.

”Eq”-noodi palauttaa joko 0 tai 1, riippuen siitä onko sille annetut arvot samoja. ”Cond”-noodi käsittelee näitä ”truthy” 0 ja 1 arvoja, palauttaen jommankumman sille annetuista noodeista. ”Cond”- ja ”eq”-noodeja yhdistelemällä saa rakennettua ”if..else” kaltaisia lauseita. Bottomsheetin tapauksessa, animaatiologiikka rakentuu perustalle ”*Jos vedetään..jos ei vedetä*”.

Reanimatedin noodit evaluoidaan uudestaan joka kerta kun niiden arvot muuttuvat. Käytännössä siis, koska animaatiolle annetut arvot tulevat suoraan eleenkäsittelijöiltä, kuviossa 23 näkyvää ylimmän tason ”*cond(eq(panState, State.ACTIVE))*” seuraavaa if-blokkia ajetaan joka kerta kun eleenkäsittelijöiltä tulee uudet arvot.

Se, että bottomsheet seuraa sormea saadaan siis rakennettua yksinkertaisella, kuviossa 24 näkyvällä funktiolla, jossa if-noodi palauttaa alkupisteen ja sormen liikkuman matkan summan.

```
return cond(
  eq(panState, State.ACTIVE),
  [cond(not(isPanning), [set(isPanning, 1)]), add(position, panY)],
  [
    cond(
      toggleTapAnim,
      [
        set(toggleTapAnim, 0),
        set(isPanning, 0),
        set(anchor, calculateTapAnchor(position)),
        set(springState.finished, 0),
        set(springState.time, 0),
        set(velocity, 1000),
        startClock(clock),
      ],
      [
        cond(isPanning, [
          set(isPanning, 0),
          set(springState.position, add(springState.position, panY)),
          set(panY, 0),
          set(
            anchor,
            closestAncor(add(springState.position, divide(velocity, 10)))
          ),
          set(springState.finished, 0),
          set(springState.time, 0),
          startClock(clock),
        ]),
        cond(not(springState.finished), [
          spring(clock, springState, springConfig),
        ]),
      ]
    ),
    position,
  ]
)
```

Kuvio 23. Bottomsheet animaatio Reanimatedilla

```
function followFinger(
  panState: Animated.Value<number>,
  panY: Animated.Value<number>
) {
  const startPosition: Animated.Value<number> = new Value(CLOSED_OFFSET)
  return cond(eq(panState, State.ACTIVE), [
    add(startPosition, panY)
  ], [
    //TODO: handle animating after pan
  ])
}
```

Kuvio 24. Vetoanimaation ensimmäinen osa Reanimated noodeilla

5.7.4 Kellot ja vetämisen jälkeinen animoiminen

Kellot

Else-blockin evaluoiminen alkaa heti, kun “panState”-muuttujan arvo muuttuu State.ACTIVE-tilasta, eli kun sormi nostetaan ruudulta.

Kappaleessa 5.7.3 mainittiin, että ifNode päivitettiin joka kerta kun eleenkäsittelijöiltä tuli uusi arvo, joka johti siihen, että sheet animoitui sormen liikkeen mukana. Mutta miten animaatio päivittyy sormen irti päästämisen jälkeen?

Sitä varten Reanimatedissa on olemassa kellot, eli ”clock”-noodit. Kellot toimivat jokaisen “itsenäisen” animaation ajurina. Itsenäisellä animaatiolla tarkoitetaan animaatiota, jonka ajurina ei toimi jokin ele.

Kello on pohjimmiltaan Animated.Value, johon päivittyy käynnissä ollessaan viimeisimmän ruudunpäivityksen aikaleima. Koska kello sisältää viimeisimmän ruudunpäivityksen aikaleiman, peräkkäisiä aikaleimoja vertailemalla saadaan laskettua ruudunpäivitysten välinen aika. Ruudunpäivitysten välistä aikaa käyttämällä voidaan laskea missä vaiheessa animaation X pitäisi olla, jotta animaatio saadaan pidettyä ajallaan mahdollisista suorituskykyongelmista huolimatta.

Kello on olennainen osa Reanimatedin animaatioita, ja siksi jokainen Reanimatedin tarjoama animaatiometodi ottaakin sen vastaan, tila- ja konfiguraatio-objektin lisäksi.

ElseNoden animaatio

Kuviossa 23 nähdään, että elseNode on todella paljon isompi kuin ifNode. Suurin osa siitä on kuitenkin animaation konfiguroimista (ks. Kuvio 25). Olennaisin osuus elseNodesta on ”*startClock(clock)*”, ja kuviossa 26 näkyvä spring()-metodin kutsu, ja sitä seuraava position arvon palauttaminen.

Yksinkertaistettuna, vetoliikkeen loppumisen sekä sheetin headerin painamisen jälkeen tapahtuu seuraava sarja tapahtumia:

- Vain kerran:
  - Edellisen animaation tila nollataan
  - Animaatiolle annetaan uusi konfiguraatio
  - ElseNoden uudelleen evaluoimiseen johtava kello laitetaan käyntiin
- Joka kerta kun elseNode evaluoidaan:
  - Jos animaatio on vielä kesken, päivitetään position-arvoa spring()-metodia käyttämällä
  - Palautetaan päivitetty position arvo

```
//#region set up post-pan animation
cond(isPanning, [
  set(isPanning, 0),
  set(springState.position, add(springState.position, panY)),
  set(panY, 0),
  set(
    anchor,
    closestAncor(add(springState.position, divide(velocity, 10)))
  ),
  set(springState.finished, 0),
  set(springState.time, 0),
  startClock(clock),
]),
//#endregion set up post-pan animation
```

Kuvio 25. Reanimated animaation konfiguroiminen

```
        cond(not(springState.finished), [
          spring(clock, springState, springConfig),
        ]),
      ]
    ),
    position,
  ]
)
```

Kuvio 26. elseNode:n animoiva osuus

6 Tekniikoiden vertaileminen

Työn React Native -projekti alustettiin Expo-viitekehystä käyttämällä, joka mahdollisti sovelluksen julkaisemisen Expon palvelimille. Täten, sovelluksen asentaminen

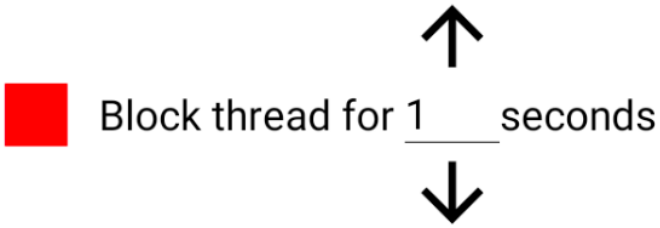
Android-laitteille onnistui skannaamalla kuviossa 27 näkyvä QR-koodi Expo sovelluksessa.

Animaatiomenetelmiä vertailtiin fyysisillä Android-laitteilla, Android-emulaattorilla ja iOS-simulaattorilla. Emulaattoritestaus suoritettiin MacBook Pro 16” tietokoneella.

Expo käynnistettiin ”—no-dev”-parametrin kanssa, jolloin emulaattoreihin asennettiin production-versio hitaamman debug-version sijaan.

Kuvio 27. Expo projektin latauskoodi

Vertailemisen avuksi luotiin ThreadBlocker-komponentti, jota käyttämällä JavaScript-thread voitiin lukita määritetyksi ajaksi. ThreadBlocker-komponentilla voitiin simuloida JavaScript puolella tapahtuvia raskaita operaatioita. (ks. Kuvio 28)



Kuvio 28. ThreadBlocker-komponentti

6.1 LayoutAnimation API

LayoutAnimation on selkeästi kaikista yksinkertaisin, mutta samalla, se on vaihtoehdoista rajoittunein. Sillä onnistuu klikkauksen animoiminen, muttei vetoliikkeen animoiminen. Se, riittääkö pelkkä klikkaamalla avaaminen sovelluksen X tarpeisiin, on määrittelykysymys.

Koska LayoutAnimationin animaatio hoidetaan käyttöliittymäpuolella, JavaScript-threadin jäätyminen ei vaikuta animaatioon sen aloittamisen jälkeen. Koska LayoutAnimation on kuitenkin riippuvainen siitä, että animaatio aloitetaan JavaScript puolelta, on hyvinkin mahdollista, että animaatio alkaa viiveellä.

6.2 Animated

Animated-kirjasto tarjoaa paljon enemmän kontrollia LayoutAnimationiin verrattuna. Plussaa Animated kirjasto saa helppokäyttöisyydestään.

Animated-kirjaston interpolaatiot mahdollistavat sen, että yhtä Animated.Valuea voidaan jatkaa moneen eri tarkoitukseen. Esimerkkinä, bottomsheet komponentissa voisi olla yksi Animated.Value, jota ajettaisiin arvojen 0 ja 1 välillä. Kyseisestä arvosta saataisiin helposti interpoloitua arvo avatulle ja suljetulle offsetille (esim. 0px ja 750px). Samasta arvosta voitaisiin myös interpoloida taustan väri, joka muuttuisi valkoisesta mustaan animaation edetessä. Kaikki animaatiot saataisiin ajettua vain yhtä arvoa muuttamalla.

Animated on kuitenkin rajoittunut JavaScript puolelle, edes useNativeDriver ei saa sitä pelastettua. Vaikka kaikki animaation ajamislogiikka ja eleen käsittelylogiikka saataisiinkin siirrettyä käyttöliittymäpuolelle, animaation käynnistämiskäskyn pitää aina kulkea JavaScript puolelta, sillan läpi käyttöliittymäpuolelle.

ThreadBlocker-komponentti auttoi demonstroimaan useNativeDriver-parametrin vaikutuksen. Esimerkissä, jossa useNativeDriver oli päällä, bottomsheetin vetäminen onnistui siitä huolimatta vaikka ThreadBlocker oli aktiivinen. Kun sheetistä päästettiin irti, sheet jäi paikoilleen odottamaan, että JavaScript puoli aloittaa animaation päämäärän laskemisen.

Sen sijaan esimerkissä jossa useNativeDriver oli poissa päältä, sheetin vetäminen ei onnistunut kun ThreadBlocker-komponentti oli aktiivinen.

Android-emulaattorilla ja hieman hitaammalla puhelimella (Samsung A50) testatessa vetoliikkeen ja määränpäähän animoitumisen välissä oli havaittavissa pieni nykäisy. Samaa nykäisyä ei ollut havaittavissa iOS-simulaattorin eikä Samsung Galaxy S10-testipuhelimen kanssa.

6.3 Reanimated

Reanimatedilla toteutettu bottomsheet pyöri jokaisella testatulla laitteella natiivia vastaavalla suoritustasolla. Reanimatediin ei vaikuttanut JavaScript-threadin lukkiutumiset. Vaikka ThreadBlocker aktivoitiin, sheet animoitui oikein haluttuun päämäärään.

ThreadBlockerin vaikutus huomattiin vasta siitä, että natiivipuolelta kutsuttava callback joka määrittää onko bottomsheetin sisällön ScrollView lukittu vai ei, ei päässyt päivittämään ScrollViewtä vasta kun ThreadBlocker oli suorittanut tehtävänsä loppuun.

7 Johtopäätökset

React Nativen natiiviuden mahdollistava silta on suorituskyvyn pullonkaula, koska sen läpi kulkeva data pitää serialisoida molempiin suuntiin kuljettaessa. Vetoliikkeet ovat todella vaativia juuri sen takia, että data kulkee JavaScript- ja käyttöliittymä-threadin välillä molempiin suuntiin. Monet nykyaikaiset käyttöliittymäkomponentit ovat kuitenkin suunniteltu eleet edellä, jolloin ainoaksi vaihtoehdoksi jää eleiden käsittelyn ja animaation siirtäminen kokonaan käyttöliittymäpuolelle.

React-native-reanimated on tällä hetkellä ainoa vaihtoehto, joka mahdollistaa nykyaikaisten, elepohjaisten ja animoitujen käyttöliittymäkomponenttien luomisen yhtä koodikantaa käyttäen.

Reanimatedin käyttö tuo mukanaan myös muita hyötyjä. Koska Reanimatedin ei tarvitse kommunikoida JavaScript puolen kanssa, silta sekä JavaScript puoli säästyvät animaation tuomalta kuormitukselta, jolloin sovelluksen toiminnalle olennaiset käskyt nauttivat vapaasta kaistanleveydestä, nopeuttaen koko sovelluksen toimintaa.

Reanimated on kuitenkin paljon vaikeampi oppia, kuin muut menetelmät. Jos bottomsheet-komponentilta olisi vaadittu vain klikkaamalla avaaminen ja sulkeminen, LayoutAnimation Api tai Animated-kirjasto olisi ollut enemmän kuin sopiva valinta tehtävää varten.

Tilanne voi kuitenkin muuttua lähitulevaisuudessa, sillä React Nativeen on tulossa isoja arkkitehtuurillisia muutoksia.

React Native tulevaisuudessa

React Native tiimi ja yhteisö on vuodesta 2018 saakka koodannut isoja arkkitehtuurillisia muutoksia React Nativeen. Muutoksien tarkoituksena on nopeuttaa JavaScriptin ja natiivipuolen kommunikointia, täten, parantaen suorituskykyä. (Sciandra 2019.)

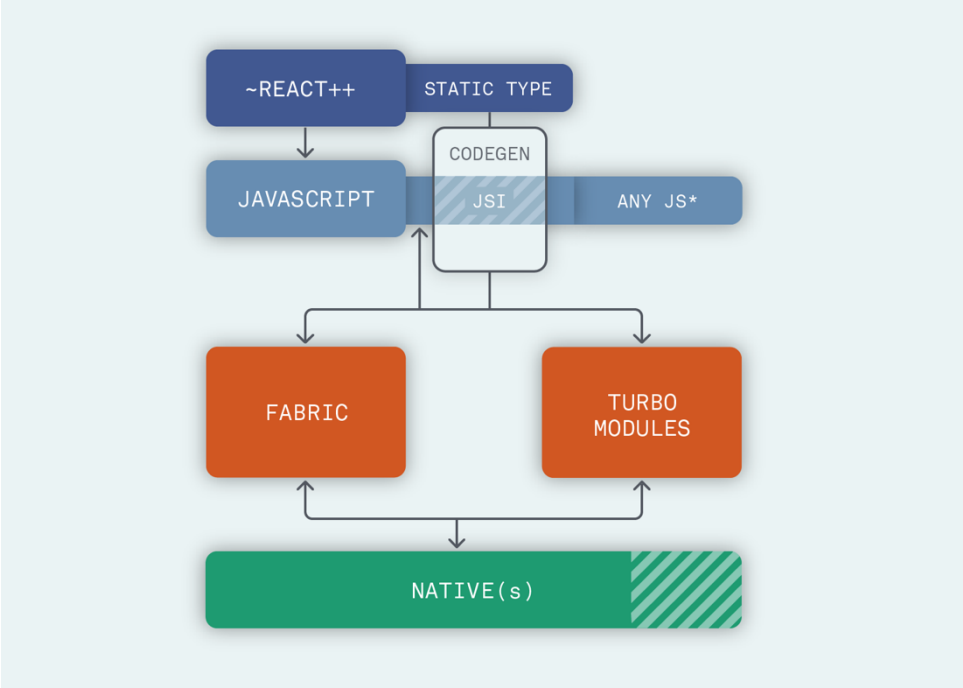
Muutosten jälkeen JavaScript puolen pitäisi pystyä kutsumaan natiivi (C++)-puolta synkronisesti JSI:n, JavaScript Interfacen, läpi. Se, että kutsuja ei enää tarvitse lähettää asynkronisesti sillan läpi, eikä validoida joka välissä kiitos uuden staattisen tyypityksen, vaikuttaa todella lupaavalta.

JSI mahdollistaa myös teoriassa minkä tahansa JavaScript moottorin käyttämisen React Nativen JavaScript koodin ajamiseen, jolloin React Nativen ei tarvitsisi paketoida JavaScriptCorea Android sovellusten mukaan (Sciandra 2019). React Nativella rakennetut Android-sovellukset voisivat hyötyä siitä, että ne saisivat Chrome-selaimen V8-JavaScript-moottorin käyttöönsä JavaScriptCoren sijaan.

Muutoksiin sisältyy myös uudistettu, tehokkaampi React, sekä uusi käyttöliittymämanageri, Fabric, joka vastaa Reactin lähettämien käyttöliittymän päivitys käskyjen lähettämisestä natiivipuolelle.

Käytännössä siis, uuden arkkitehtuurin on tarkoituksena höylätä kaikki hidastavat tekijät pois React Nativen sisäisestä toiminnasta. Uuden arkkitehtuurin karkea hahmotelma näkyy kuviossa 29.

Uusi arkkitehtuuri voi mahdollistaa sen, että Animated kirjaston suorituskyky pääsee niin lähelle Reanimatedin suorituskykyä, ettei Reanimatedin opetteleminen ole enää kannattavaa.



Kuvio 29. React Nativen uusi arkkitehtuuri (Sciandra, 2019)

Lähteet

42matters. 2019. Distribution of free and paid apps in the Apple App Store and Google Play as of December 2019. Viitattu 01.02.2020. https://www-statista-com.ezproxy.jamk.fi:2443/statistics/263797/number-of-applications-for-mobile-phones/

Appfigures & VentureBeat. 2020. Number of apps available in leading app stores as of 4th quarter 2019. Viitattu 01.02.2020. https://www-statista-com.ezproxy.jamk.fi:2443/statistics/276623/number-of-apps-available-in-leading-app-stores/

Gaba, R & Ramachandran, A. 2018. React Made Native Easy. Viitattu 15.02.2020. https://www.reactnative.guide/

Developer Survey Results. 2019. Verkkosivu Stackoverflow www-sivuilla. Viitattu 12.01.2020. https://insights.stackoverflow.com/survey/2019

Introduction to the DOM. 2020. Dokumentaatio Mozilla Web Docs www-sivuilla. Viitattu 01.02.2020. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

JNI tips. 2020. Dokumentaatio Android Developers foorumilla. Viitattu 08.02.2020. https://developer.android.com/training/articles/perf-jni#general-tips

Newzoo. 2019. Number of smartphone users worldwide from 2016 to 2021 (in billions). Viitattu 01.02.2020. https://www-statista-com.ezproxy.jamk.fi:2443/statistics/330695/number-of-smartphone-users-worldwide/

Reconciliation. 2019. Dokumentaatio ReactJS www-sivuilla. Viitattu 01.02.2020. https://reactjs.org/docs/reconciliation.html

Sciandra, L. 2019. The New React Native Architecture Explained: Part 1-4. Kokoelma blogipostauksia Formidable www-sivuilla. Viitattu 01.01.2020. https://formidable.com/blog/2019/react-codegen-part-1/

StatCounter. 2020. Mobile operating systems' market share worldwide from January 2012 to December 2019. Viitattu 01.02.2020. https://www-statista-com.ezproxy.jamk.fi:2443/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/