Tampereen ammattikorkeakoulu

Reaaliaikaisen full stack -sovelluksen kehittäminen web-tekniikoin

Heidi Häti

OPINNÄYTETYÖ
Syyskuu 2019

Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotanto

HÄTI, HEIDI
Reaaliaikaisen full stack -sovelluksen kehittäminen web-tekniikoin

Opinnäytetyö 38 sivua
Syyskuu 2019

Reaaliaikaisuuden toteuttaminen web-tekniikoin tuo omat haasteensa sovelluskehitykseen, koska toimivuuteen tarvitaan kaksi itsenäistä sovellusta, palvelin ja asiakasohjelma. Opinnäytetyön tavoitteena oli selvittää, kuinka reaaliaikaisuuden toteutus voidaan saavuttaa sovelluksessa web-tekniikoin. Tarkoituksena oli tutustua eri menetelmiin, joilla voidaan luoda reaaliaikaisia sovelluksia ja kehittää yksinkertainen sovellus näitä tekniikoita hyödyntäen.

Opinnäytetyön tuotoksena kehitettiin React Native sekä Node.js teknologioita hyödyntäen mobiilisovellus, joka toimii reaaliajassa WebSocket protokollaa hyödyntäen. Sovelluksella eri käyttäjät voivat keskustella toistensa kanssa lähettämällä viestejä.

Tuloksista nähdään, että kehittäjien tulisi suosia push-tekniikoita, kuten Server-Sent Events ja WebSocket pull-tekniikoiden sijaan. Push-tekniikat kuormittavat palvelinta huomattavasti enemmän, jonka takia ne eivät sovellu käsittelemään nykyajan käyttäjämääriä.

Asiasanat: full stack, reaaliaikaisuus, web-tekniikat

ABSTRACT

Tampere University of Applied Sciences
Business Information Systems
Software Production

Heidi Häti
Developing a real time full stack application using web technologies

Bachelor's thesis 38 pages
September 2019

Creating an application that works in real time using web technologies brings its own set of problems to the development, because for the application to work, two different independent software are needed: server and client. The purpose of the thesis was to figure out different techniques how real time can be achieved in a full stack application. The objective was to familiarize with different methods that real time applications can be developed, and create a simple application using these methods.

As a result of the thesis, a mobile application was developed using React Native and Node.js technologies, which works in real time using WebSocket protocol. With the applications, users can chat with each other by sending messages.

The results indicate that developers should use push techniques, such as Server-Sent Events and WebSocket, instead of pull techniques. Push techniques strain the server significantly more, and because of that, they are not suitable to handle modern amounts of users.

Key words: full stack, real time, web techniques

SISÄLLYS

ERITYISSANASTO

Asiakas	Laite tai sovellus, joka on yhteydessä palvelimeen
Palvelin	Ohjelma tai laite, joka antaa toiminnallisuuksia asiakkaalle
Pull-tekniikka	Web-tekniikka, jossa asiakas hakee tietoa palvelimelta
Push-tekniikka	Web-tekniikka, jossa palvelin lähettää tietoa suoraan asiakkaille
AJAX	Asynchronous JavaScript and XML: kokoelma teknologioita, jotka mahdollistavat asynkronisien web-sovellusten toteuttamisen
JSON	JavaScripts Object Notation: Yksinkertainen avoimen standardin tiedostomuoto tiedonvälitykseen
Protokolla	Käytäntö tai standardi, joka määrittelee tai mahdollistaa laitteiden tai ohjelmien väliset yhteydet
TCP	Transmission Control Protocol: Tietoliikenneprotokolla, jolla luodaan yhteyksiä tietokoneiden välille
P2P	Peer to peer: verkko jossa ei ole kiinteitä palvelimia ja asiakkaita, vaan jokainen verkkoon kytketty taho toimii sekä palvelimena että asiakkaana verkon muille jäsenille.
Singleton	Luokka, joita on vain yksi instanssi
Cross-platform	Alustariippumaton sovellus
Natiivisovellus	Sovellus, joka on kehitetty tietylle alustalle
ORM	Object-relational mapping: Tekniikka datan muuttamiseen yhteensopimattomien järjestelmien välillä

1 JOHDANTO

Reaaliaikaisella systeemillä tarkoitetaan, että se ei toimi pelkästään loogisesti oikein, vaan myöskin tietyn ajan sisällä. Reaaliaikaisen systeemin toiminnalle voidaan antaa aikaraja, jossa sen operaatiot on suoritettava. (Shin & Ramanathan 1994) Hyvä esimerkki sovelluksesta, joka toimii reaaliajassa on Skype. Käyttäjät voivat lähettää toisilleen viestejä, tai soittaa videopuheluita. Kaikki käyttäjät, jotka ovat puhelussa voivat keskustella keskenään sujuvasti, koska video ja ääni liikkuvat verkossa reaaliajassa.

Reaaliaikaisuuden saavuttamien vaatii, että uusi tai muuttunut tieto saavuttaa käyttäjän mahdollisimman nopeasti. Full stack viittaa ohjelmistokokonaisuuteen, joka tarvitaan sovelluksen toimivuuteen. (Oxford) Yleensä full stack -sovellus koostuu palvelimesta, asiakasohjelmasta sekä tietokannasta, joiden tulisi kommunikoida saumattomasti keskenään, jotta viive saataisiin niin alhaiseksi, että sovellus voisi saavuttaa reaaliaikaisuuden. Tekniikat, joilla palvelin ja asiakas kommunikoivat keskenään voidaan jaotella pull- ja push-tekniikoihin. Pull-tekniikoissa asiakas on vastuussa tiedonhausta, ja kyselee palvelimelta mahdollisista muutoksista. Push-tekniikoissa palvelin lähettää muutokset asiakkaalle, heti niiden ilmettyä.

Tässä työssä tutustutaan eri web-tekniikoihin, joilla full stack -sovelluksen reaaliaikaisuus voidaan saavuttaa. Reaaliaikainen web on ollut olemassa jo pitkään, ja tekniikoita, joilla se voidaan saavuttaa, on tullut useita vuosien saatossa. Sen suosio on kasvanut ajan myötä, kun tekniikat ovat kehittyneet, jonka ansiosta voidaan toteuttaa teknisesti vaativampia sovelluksia.

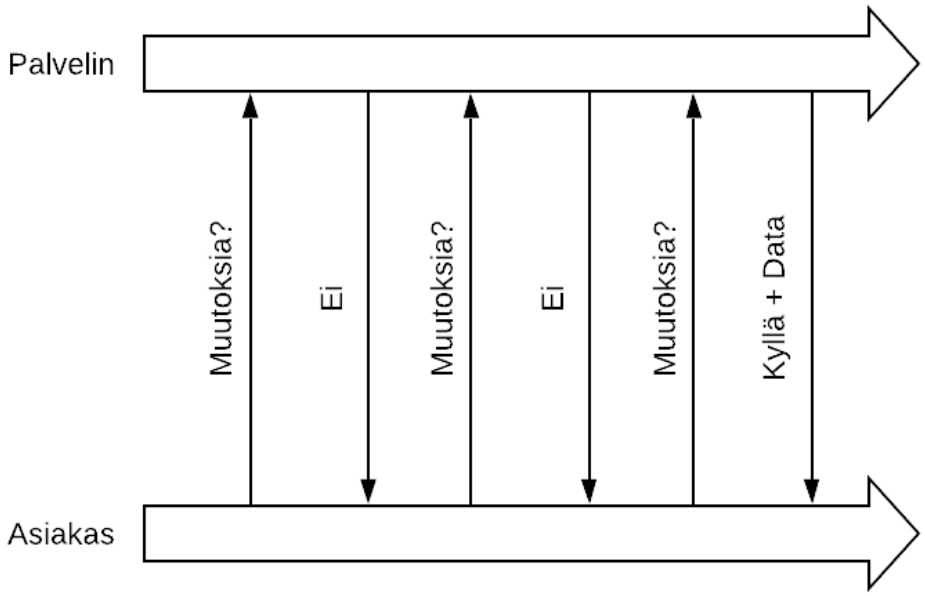
Työn tarkoitus on toteuttaa prototyyppi full stack -sovelluksesta, joka toimii reaaliajassa. Sovelluksella eri käyttäjät voivat keskustella keskenään lähettämällä viestejä. Prototyypissä käytetyt tekniikat antavat kokonaiskuvaa siitä, miten reaaliaikaisia sovelluksia kehitetään, ja mitä tulee ottaa huomioon. Tekniikoita soveltamalla voidaan kehittää huomattavasti vaativampia ja monimutkaisempia ohjelmistoja.

2 REAALIAIKAISET WEB-TEKNIIKAT

2.1 Http-polling

Http-polling -tekniikassa, joka tunnetaan myös nimellä short polling, asiakas hakee uutta, tai päivittynyttä tietoa palvelimelta. Koska tiedonhaku on asiakkaan tehtävä, http-polling luokitellaan pull-tekniikaksi. Asiakkaalla ei ole tietoa palvelimen tapahtumista, joten sen on lähetettävä AJAX-kysely mahdollisista muutoksista. Jotta sovelluksen reaaliaikaisuus voitaisiin saavuttaa, kyselyjä on lähetettävä jatkuvasti, vaikka muutoksia ei välttämättä olisikaan. (Loreto, Saint-Andre, & Salsano 2011)

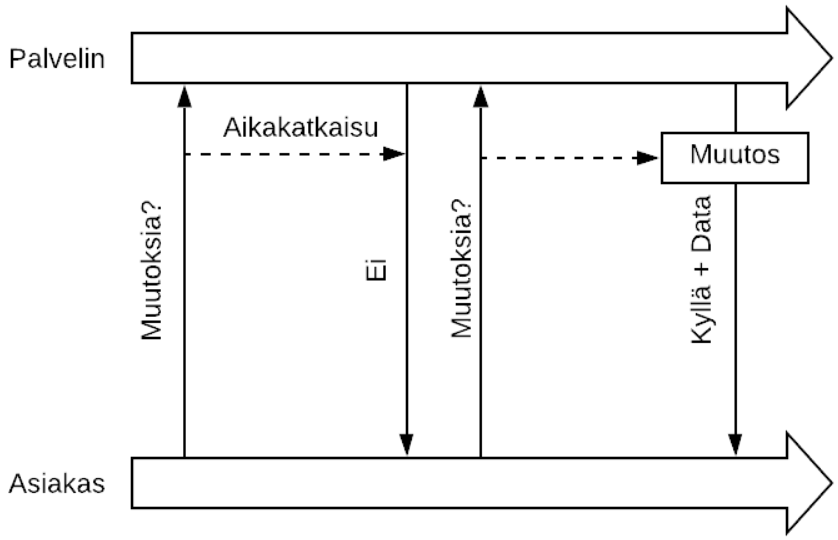
Jatkuva kyselyiden lähetys kuormittaa palvelinta turhaan, varsinkin jos käyttäjiä on suuri määrä. Palvelin joutuu käsittelemään dataa, vaikka mitään lähetettävää ei olisi. Palvelimen ylikuormitusta voidaan ehkäistä nostamalla aikaa, jonka välein kyselyjä lähetetään. Jos kyselyjä kuitenkin lähetetään liian harvoin, päivityksiä saattaa jäädä välistä. Kuviossa 1 nähdään http-polling toimintaperiaate. Mainituista syistä http-polling ei ole suositeltu tekniikka, varsinkaan suurissa sovelluksissa, vaikka se on vakaa ja helppo ratkaisu. Tekniikka ei yksinkertaisesti skaalaudu tarpeeksi hyvin nykyisten massiivisten käyttäjämäärien käsittelyyn.



KUVIO 1. Asiakas lähettää kyselyn palvelimelle mahdollisista muutoksista

2.2 Long-polling

Long-polling on variaatio perinteisestä http-polling -tekniikasta. Aivan kuten http-pollingissa, asiakas lähettää pyynnön palvelimelle. Palvelin ei kuitenkaan lähetä tyhjää vastausta, jos muutoksia ei ole, vaan jättää pyynnön auki ja jää odottamaan uutta tietoa tai aikakatkaisua. Kun palvelimella on uutta tietoa, se lähettää heti http-vastauksen asiakkaalle, joka sulkee alkuperäisen pyynnön. Yleensä asiakas lähettää vastauksen saatuaan heti uuden pyynnön palvelimelle, jolloin päivitysten viive saadaan lähes kokonaan eliminoitua. Long-pollingia ei voi luokitella push-tekniikaksi, koska asiakas on vastuussa yhteydenotosta. Sillä voidaan kuitenkin jäljitellä palvelimelta tiedon puskemista tilanteissa, joissa palvelimelta ei pystytä lähettämään tietoa ilman kyselyä. Kuviossa 2 kuvataan long-polling tekniikkaa.



KUVIO 2. asiakas lähettää kyselyn muutoksista ja palvelin vastaa muutosten tapahduttua

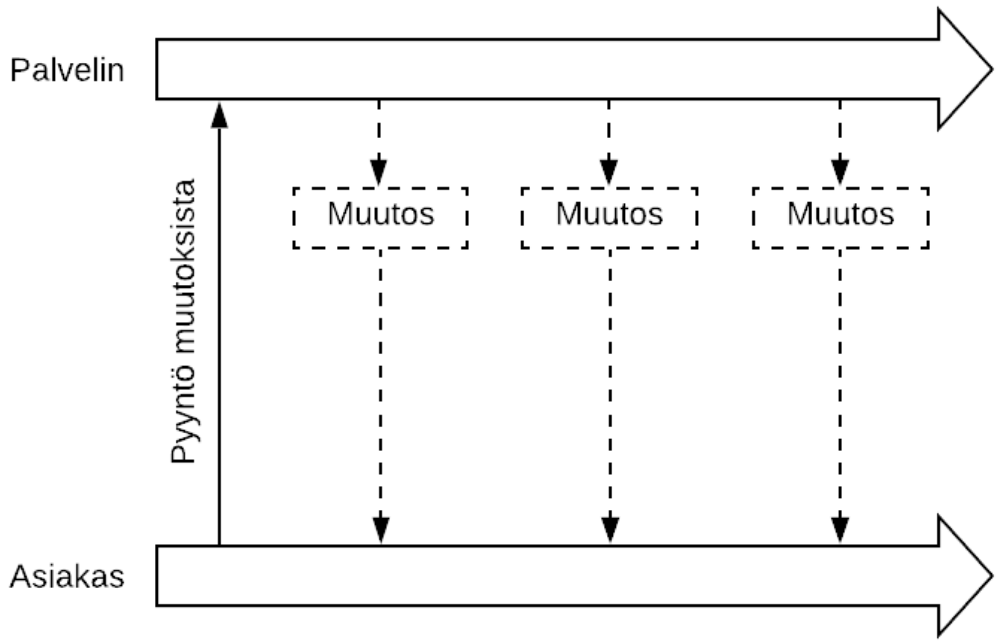
Long-pollingin avulla haluttiin korjata perinteisen http-pollingin ongelmia, kuten esimerkiksi rajoittaa lähetettyjen kyselyjen määrää ja parantaa sovellusten skaalautuvuutta. Kyselyiden auki jättäminen tuo kuitenkin omat ongelmansa kehittämiseen. Tilanteesta riippuen palvelin saattaa joutua käsittelemään kyselyt esimerkiksi erillisissä säikeissä. Koska monien säikeiden käsittely kuluttaa muistia,

niitä voidaan pitää käynnissä vain rajattu määrä. Kun määrä on täysi palvelin ei enää voi ottaa vastaan uusia kyselyjä. (Genchev 2008)

Koska long-polling -toiminnan pohja on AJAX, se toimii lähes kaikissa laiteissa ja selaimissa. AJAX on kokoelma teknologioita, joiden avulla web-sovellukset voivat tehdä päivityksiä käyttöliittymään lataamatta koko web-sivua uudestaan. (MDN web docs AJAX 2019) Kehityksessä ei ole siis yleensä tarvetta huolehtia yhteensopivuusongelmista. Long-pollingin tuomat ongelmat palvelinohjelmistoissa kuitenkin rajoittavat sen käyttöä vain pieniin projekteihin. (Loreto, Saint-Andre & Salsano 2011)

2.3 Server-Sent Events

Toisin kuin edellä mainituissa tekniikoissa, Server-Sent Events eli *SSE* -tekniikassa palvelin on vastuussa tiedon lähettämisestä asiakkaalle. SSE on yksisuuntainen kommunikointimenetelmä, jossa vain palvelin lähettää tietoa, joten se luokitellaan push-tekniikaksi. Asiakas avaa yhteyden palvelimelle ja jää kuuntelemaan mahdollisia muutoksia. Palvelin ei kuitenkaan voi kuunnella asiakasta samalla tavalla. (Hickson 2015) Kuviossa 3 kuvataan SSE -tekniikan toimintaa.



KUVIO 3. Asiakas avaa yhteyden palvelimelle ja kuuntelee muutoksia

Palvelin lähettää asiakkaalle *viestejä*. Viestit voivat sisältää attribuutit ID, event ja data. ID on viestin tunniste, jota voidaan hyödyntää esimerkiksi virhetilanteissa. Event on viestin tyyppi. Koska palvelin voi lähettää erityyppisiä viestejä, asiakas päättää tilanteesta riippuen, miten saapuvat muutokset käsitellään. Data on nimensä mukaisesti palvelimen konkreettiset muutokset, ja ainoa attribuutti, joka on pakko sisällyttää lähetettävään viestiin. Data on aina tekstiä, mutta sitä voidaan jaotella useammalle riville, jolloin voidaan luoda esimerkiksi rakenteeltaan monimutkaisempia JSON-dataa. (JavaScript.info Server Sent Events)

Jos verkkoyhteys katkeaa, *EventSource* käynnistää virhetapahtuman, joka yrittää automaattisesti yhdistää palvelimelle (MDN web docs EventSource 2019). Uudelleenyhdistämisen yhteydessä voidaan tarkistaa asiakkaan viimeksi saaman tapahtuman ID, jolloin palvelin tietää mitä tietoa asiakas ei ole saanut katkoksen takia. Kun yhteys on jälleen muodostettu, välistä jääneet tapahtumat voidaan lähettää asiakkaalle. (JavaScript.info Server Sent Events)

SSE-tekniikka mahdollistaa reaaliaikaisen sovelluksen kehittämisen, koska palvelin ei jää odottamaan asiakkaan pyyntöä, ennen kuin se puskee uuden tiedon eteenpäin. Lisäksi palvelin ei joudu käsittelemään monen asiakkaan jatkuvia pyyntöjä, mikä edesauttaa sovelluksen skaalautuvuutta, koska palvelin pystyy käsittelemään suuriakin käyttäjämääriä kerrallaan. Jokaisella asiakkaalla tulisi olla palvelimelle vain yksi yhteys, joka muodostetaan tapahtumien kuuntelun alkaessa. SSE soveltuu hyvin projekteihin, joissa palvelimelta lähetetään paljon muutoksia, kuten esimerkiksi sosiaalisen median syötteet. Se ei kuitenkaan ole hyvä ratkaisu tilanteisiin, jossa asiakkaan tulisi pystyä lähettämään tehokkaasti muutoksia palvelimelle. Koska SSE on yksisuuntainen kommunikointimenetelmä, asiakas ei voi lähettää viestejä palvelimelle. Jos asiakkaasta halutaan lähettää muutoksia, on turvauduttava esimerkiksi AJAX-pyyntöihin.

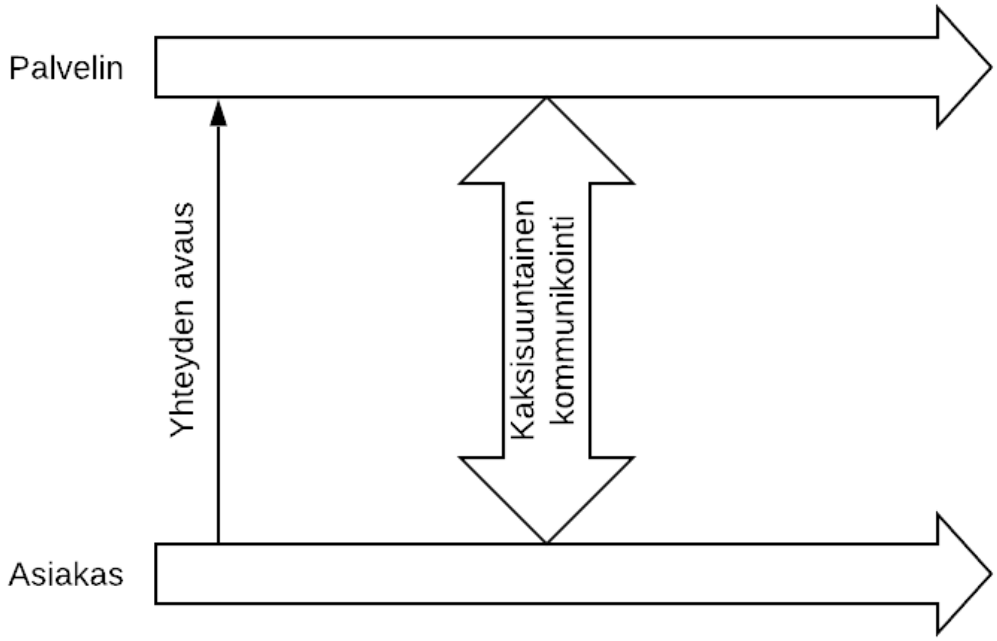
2.4 WebSockets

WebSocket on protokolla, joka tukee kaksisuuntaista kommunikointia palvelimen ja asiakkaan välillä. Kommunikointi tapahtuu yhden avonaisen TCP yhteyden

avulla, jonka asiakas avaa tavallisen pyynnön avulla. Tietoa siirretään viestien avulla, joita molemmat, palvelin ja asiakas, voivat lähettää. Tämän ominaisuus johtaa minimaaliseen viiveeseen sovelluksessa. (Fette & Melnikov 2011)

WebSocketit omaavat neljä tapahtumaa: *open*, *message*, *close* ja *error*. Kun yhteys asiakkaan ja palvelimen välillä on muodostettu, open tapahtuma laukaistaan. Open-tapahtumaa voidaan käyttää esimerkiksi asiakkaan ja palvelimen tietojen alustamiseen. Message-tapahtuma laukaistaan, kun palvelin lähettää tietoa asiakkaalle. Palvelin voi lähettää tekstiä, binääridataa tai kuvia. Close-tapahtuma merkkaa asiakkaan ja palvelimen yhteyden katkaisua. Yhteyden katkaisun jälkeen viestejä ei voi enää lähettää. Error-tapahtuma laukaistaan aina virheen tapahtuessa. Error-tapahtumaa seuraa aina close-tapahtuma, joka katkaisee yhteyden palvelimen ja asiakkaan välillä. Asiakas voi kommunikoida palvelimen kanssa send- ja close-toiminnoilla. Palvelimelle voidaan lähettää tietoa send toiminnon avulla, AJAX-pyynnön sijaan. Aivan kuten palvelin, asiakas voi lähettää tekstiä, binääri dataa sekä kuvia. Close-toiminnolla voidaan sulkea yhteys, jonka jälkeen dataa ei voi siirtää, ennen kuin yhteys jälleen avataan. (Tutorial points)

Kaksisuuntaisen kommunikoinnin ansiosta WebSocketit soveltuvat suuriin ja vaativiin projekteihin, joissa liikkuu paljon dataa asiakkaan ja palvelimen välillä. WebSocket-teknologian avulla voidaan luoda esimerkiksi ääni- tai videochat sovelluksia, tai verkossa toimivia moninpeleja, joissa palvelimen ja asiakkaan kommunikoinnin reaaliaikaisuus on välttämätöntä sovelluksen toiminnan kannalta. (JavaScript.info. WebSocket) Samankaltainen ”*todellinen reaaliaikaisuus”* ei ole toteutettavissa muilla mainitsemillani tekniikoilla. Kuviossa 4 nähdään WebSocket-tekniikan toimintaperiaate.



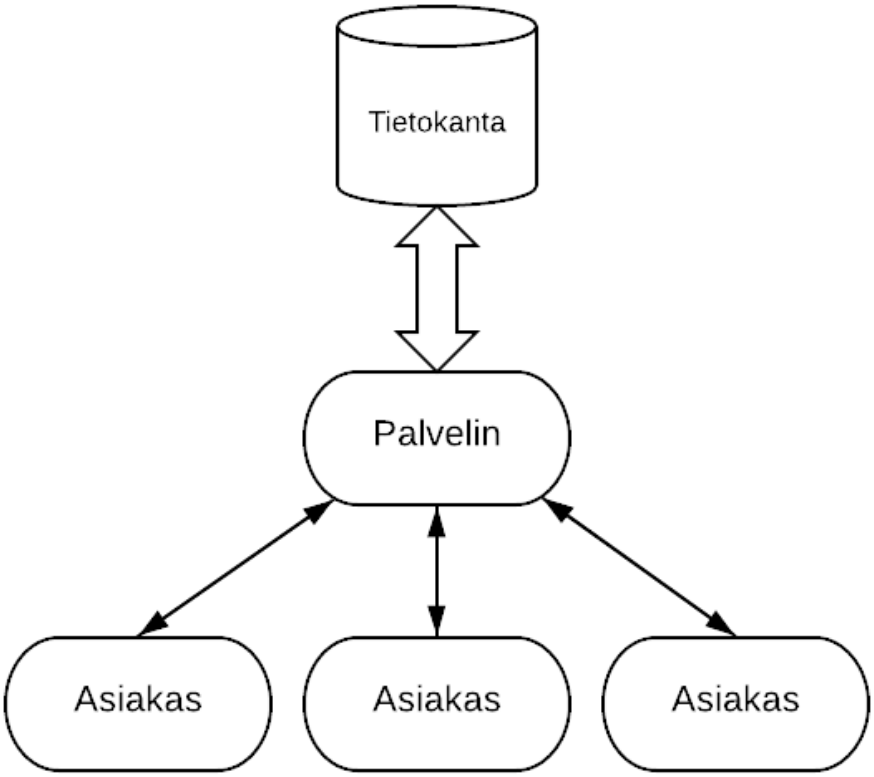
KUVIO 4. Asiakas ottaa yhteyden palvelimeen ja avaa kaksisuuntaisen kommunikoinnin

3 SOVELLUKSEN RAKENNE

Opinnäytetyön tarkoituksena on rakentaa prototyyppi reaaliaikaisesta viestintäsovelluksesta. Sovellukseen voidaan rekisteröidä useita käyttäjiä, jotka tallennetaan tietokantaan. Lisäksi kaikki lähetetyt viestit tallennetaan, jotta käyttäjät voivat lukea aikaisempia keskustelujaan. Käyttöliittymä on Android mobiilisovellus, jonka avulla kaikki käyttäjät voivat keskustella toistensa kanssa.

Sovellus rakennetaan asiakas-palvelin -mallin mukaisesti. Sovelluksessa on yksi palvelin, johon kaikki asiakkaat ottavat yhteyden. Palvelin hoitaa kaiken tiedon käsittelyn asiakkaiden ja tietokannan välillä. Asiakkaat eivät siis ole tietoisia toisistaan, eivätkä ne voi vaihtaa tietoa toistensa kesken. Datankäsittely on hyvä tehdä palvelimessa, koska se voi tarkistaa onko asiakkaan lähettämä tieto sopivaa sovellukselle. Asiakas-palvelin -mallin avulla voidaan ennaltaehkäistä virhetilanteita ja edistää tietoturvallisuutta. Jos asiakas haluaa esimerkiksi lähettää viestin toiselle asiakkaalle, palvelimessa voidaan tarkistaa, onko viestissä kaikki tarvittavat tiedot, kuten viestin sisältö ja sen asiakkaan id, jolle viesti halutaan lähettää. Lisäksi palvelimella voidaan tarkistaa, onko asiakkaalla oikeudet tehdä kyseinen toiminta. Tietyiltä käyttäjiltä voidaan haluta estää toimintoja, kuten esimerkiksi tiedon poisto tietokannasta, tai tietyn tiedon saaminen. Asiakas-palvelin -mallin avulla on mahdollista myöskin tehdä käyttöjärjestelmä vapaa sovellus. Koska kaikki datakäsittely tehdään protokollien avulla, ainua vaatimus palvelimen käyttöön on verkkoyhteys. (Reese 2000)

Jos sovelluksella on paljon käyttäjiä, ja palvelin joutuu käsittelemään paljon kyselyjä samanaikaisesti, se saattaa ylikuormittua. Lisäksi, jos palvelin ei syytä tai toisesta toimi, sovellusta ei voida käyttää, toisin kuin esimerkiksi P2P-mallin sovelluksia, jossa resurssit on yleensä jaettu moneen eri paikkaan. Asiakas-palvelin -malli on kuitenkin huomattavasti skaalautuvampi ja helpompi ylläpitää P2P-malliin verrattuna. Palvelimia voidaan lisätä, tai niiden toimintaa voidaan muokata vaikuttamatta asiakkaan toimintaan. (ESDS 2011) Kuviossa 5 kuvataan asiakas-palvelin -mallia.



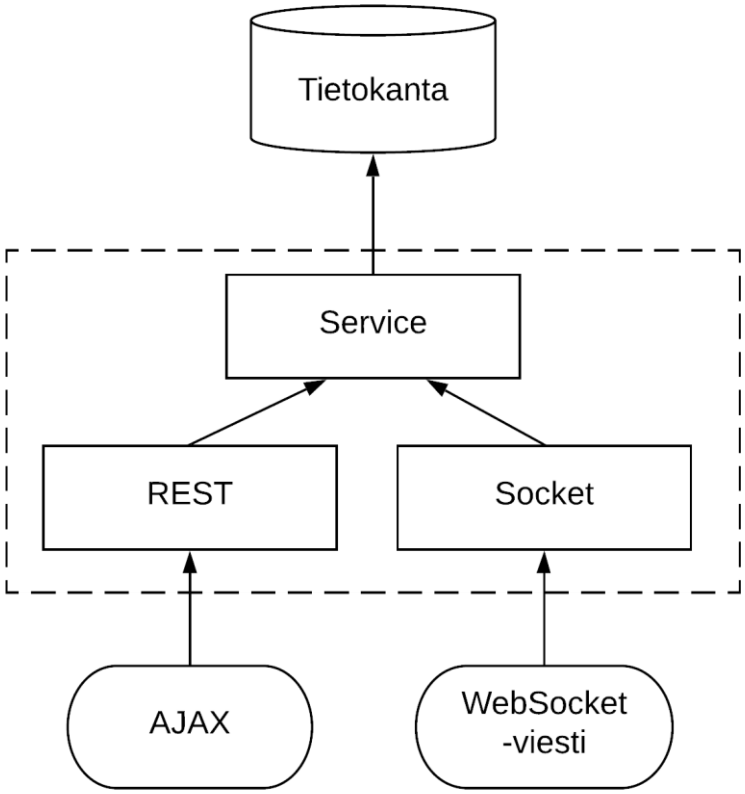
KUVIO 5 Sovelluksen rakenne

3.1 Palvelin

Palvelin käyttää WebSocket-teknologiaa, jotta päivitykset voidaan tehdä reaaliajassa. Se sisältää myös pienimuotoisen REST-rajapinnan, jolla voidaan suorittaa tiedonhakua sekä tallentaa käyttäjiä.

Palvelin voidaan jakaa kolmeen osaan: REST, Socket ja Service. Kaikki asiakkaan pyynnöt käsitellään REST-osassa. REST jaetaan *controllereihin*. Esimerkiksi kaikki käyttäjiin liittyvät pyynnöt ohjataan *userController* luokkaan, jossa pyyntö käsitellään. WebSocket-viestit käsitellään Socket osassa. Controllereiden ja Socketien toiminta on hyvin samanlaista. Suurin ero on Socket ei lähetä vastausta asiakkaalle. Pyynnön tai viestin data välitetään sille omistetulle servicelle. Esimerkiksi kaikelle käyttäjienhallintaan tarvittaville toiminoille on omistettu oma service, kuten myös viestien hallinnalle. Servicessä on tarkoitus tarkistaa, onko asiakkaalta tullut data validia sekä käsitellä data pyynnöstä riippuen. Servicet

ovat myöskin vastuussa kommunikoinnista palvelimen ja tietokannan välillä, eli tiedon lisäys, muokkaus sekä poisto. Kuviossa 6 on kuvattu palvelimen rakenne.



KUVIO 6 Palvelimen rakenne

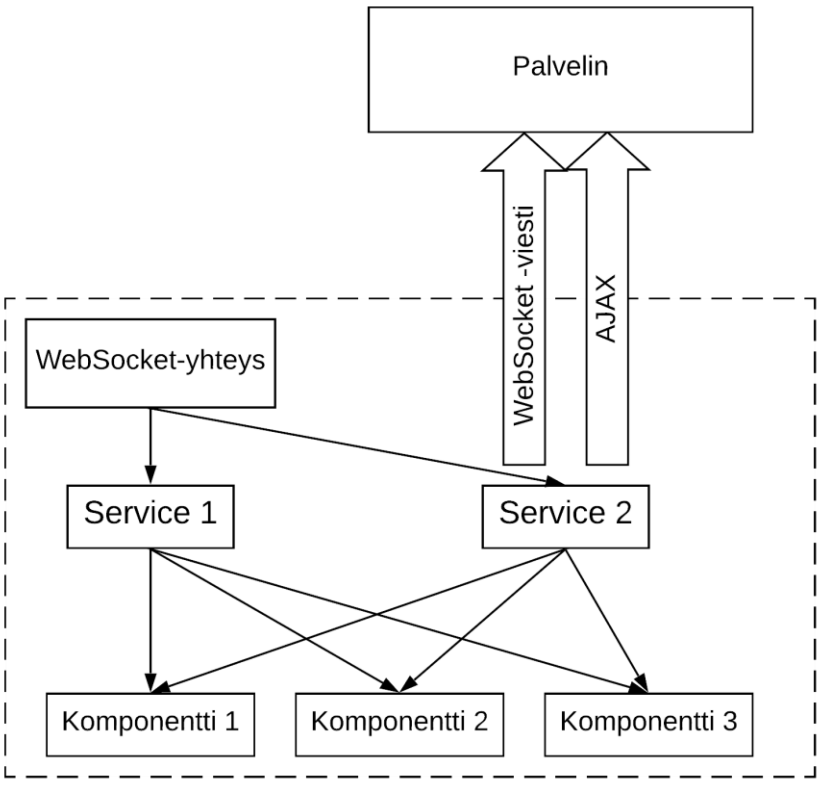
3.2 Asiakas

Asiakas on sovelluksen mobiilikäyttöliittymä. Sen avulla voidaan rekisteröidä uusia käyttäjiä, sekä lähettää viestejä toisille käyttäjille. Sovellus luodaan Android käyttöjärjestelmälle.

Asiakasohjelman käynnistyessä lähetetään pyyntö palvelimelle, jolla tarkistetaan, onko kyseinen laite tallennettu tietokantaan. Jos laitetta ei löydy, käyttäjä ohjataan näkymään, jossa valitaan nimimerkki ja lähetetään pyyntö rekisteröity-

misestä. Kun käyttäjä on tunnistettu, sovellus avaa WebSocket yhteyden palvelimelle. WebSocket yhteyksiä tulee olla vain yksi, joten yhteys tulee tallentaa singleton olioon.

Kommunikointi palvelimen kanssa tapahtuu servicen kautta. Myös servicet ovat singleton luokkia, koska monet komponentit voivat käyttää samaa serviceä samanaikaisesti. Servicet ottavat käyttöön luodun WebSocket-yhteyden, jonka avulla ne kommunikoivat palvelimen kanssa. Serviceistä voidaan lähettää AJAX-pyyntöjä, sekä lähettää ja vastaanottaa WebSocket-viestejä. Kaikki tarvittava data palvelimen tapahtumista tallennetaan serviceihin, josta komponentit voivat käyttää sitä. Komponenttien tarkoitus on luoda käyttöliittymä. Käyttäjälle halutaan näyttää lista muista käyttäjistä, josta voidaan valita keskustelukumppani. Keskustelu näkymässä ladataan ensin mahdolliset entiset viestit, joita käyttäjät ovat keskenään lähetelleet, sekä tekstikenttä, jonka avulla voidaan kirjoittaa uusia viestejä. Lisäksi käyttäjälle halutaan näyttää ilmoitus uusista viesteistä, jotka eivät tule avoinna olevasta keskustelusta. Kuviossa 7 on kuvattu asiakasohjelman rakennetta.



KUVIO 7 Asiakasohjelman rakenne

3.3 Mongo-tietokanta

Tietokanta rakennetaan Mongo-tietokannan avulla. Toisin kuin SQL pohjaisissa tietokannoissa, Mongossa ei tarvitse määritellä tauluja ennen datan tallentamista. Taulujen sijaan, data tallennetaan kokoelmiin dokumentti muodossa. Tämän ansiosta käyttöönotto on nopeaa ja helppoa, koska tietokantaa ei ole vielä välttämättä tarvinnut suunnitella ennen kehittämisen aloitusta.

Mongon ehdottomat vahvuudet ovat sen skaalautuvuus sekä suorituskyky. Se on *dynaamisesti skaalautuva* tietokanta, joka käytännössä tarkoittaa, että samassa kokoelmassa voi olla dokumentteja, joiden datarakenteet eroavat toisistaan. Tämän ansiosta Mongo soveltuu projekteihin, jossa tietokannan tarpeet saattavat yllättäen muuttua, tai sen rakennetta on vaikea määritellä selkeästi. Määrittelemätöntä skeemaa voidaan pitää myös negatiivisena asiana, sillä se saattaa johtaa epäjohdonmukaisuuksiin, sekä virhetilanteisiin. Koska tietokannasta noudettavan datan rakennetta ei aina voida ennakoida, se on otettava huomioon ohjelmoidessa.

Tietokannan dokumentit tallennetaan BSON formaatissa, eli *binary-serialized JSON*. Koska BSON on nimensäkin mukaan kehitetty JSON:in päälle, tietokannassa voidaan käyttää datatyyppejä kuten esimerkiksi listoja. Lisäksi jos tietoa käsitellään palvelimella valmiiksi JSON formaatissa, sitä ei tarvitse erikseen muuttaa tietokantaan sopivaksi, joka parantaa suorituskykyä. (Dayley 2014)

Tietokantaan tallennetaan kaikki käyttäjät, sekä lähetetyt viestit. Käyttäjä-dokumentit sisältävät attribuutit käyttäjänimi, sekä laitetunniste. Jokaisella Android-laitteella on oma uniikki laitetunniste, joten sitä voidaan hyödyntää käyttäjien tunnistamiseen. Laitetunnisteen avulla tarkistetaan esimerkiksi, onko kyseinen käyttäjä jo rekisteröitynyt sovellukseen. Viesti-dokumentteihin tallennetaan kaksi käyttäjää: lähettäjä sekä vastaanottaja. Dokumentti sisältää luonnollisesti myös viestin sisällön, eli tekstin, sekä aikaleiman, jolloin viesti on luotu.

4 TEKNOLOGIAT

4.1 React Native

Sovelluksen front-end, eli asiakasohjelma toteutetaan React Nativella. React on JavaScript kirjasto, joka on tarkoitettu web-käyttöliittymien rakentamiseen. Sen avulla voidaan luoda monimutkaisia käyttöliittymiä pienistä, itsenäisesti toimivista palasista, eli *komponenteista*. Reactilla rakennetaan *Single-Page applikaatioita*, eli web-sovelluksia, jotka lataavat vain yhden HTML-sivun, jota päivitetään dynaamisesti käyttäjän vuorovaikutuksesta. (ReactJs)

React komponentiin on tallennettu sen *tila*. Komponentin tila on JavaScript olio, johon voidaan asettaa haluttu data. Tila voidaan päivittää *setState* funktiolla, jolle annetaan parametrinä uusi tila. SetState funktion jälkeen kutsutaan automaattisesti komponentin *render* funktiota, joka piirtää komponentin. Komponenttien piirto tapahtuu tekniikalla nimeltä *JSX*, joka näyttää lähes täysin samalta kuin HTML, mutta se on syntaksi jatke JavaScriptille, jonka ansiosta kaikki JavaScriptin hyödyt on käytössä. JSX-mallin sisällä voidaan luoda toisia React-komponentteja. Näille lapsikomponenteille voidaan antaa dataa pääkomponentilta *properteina* eli props-oliona. (ReactJs. Introducing JSX; ReactJs. State and Lifecycle) Jos lapsen täytyy tehdä muutoksia annettuun dataan, se suoritetaan funktiossa, joka palauttaa pääkomponentille muutokset. Tätä kutsutaan Flux-arkkitehtuuriksi. (Flux docs)

Flux-arkkitehtuuri on toimiva ratkaisu pienissä ja suoraviivaisissa ohjelmissa, mutta voi tuoda ongelmia suuriin projekteihin. Jos lapsikomponentti tarvitsee tietoa korkealta tasolta ohjelmaa, data täytyy syöttää kaikkien välissä olevien komponenttien läpi. Tätä kommunikointi ongelmaa voidaan lievittää tekemällä luokkia, joiden tarkoitus on jakaa dataa, kuten esimerkiksi tämän opinnäytetyön tuotoksen service-luokat. Lisäksi tähän tarkoitukseen on luotu myös ulkopuolisia kirjastoja, joista tunnetuin on Redux.

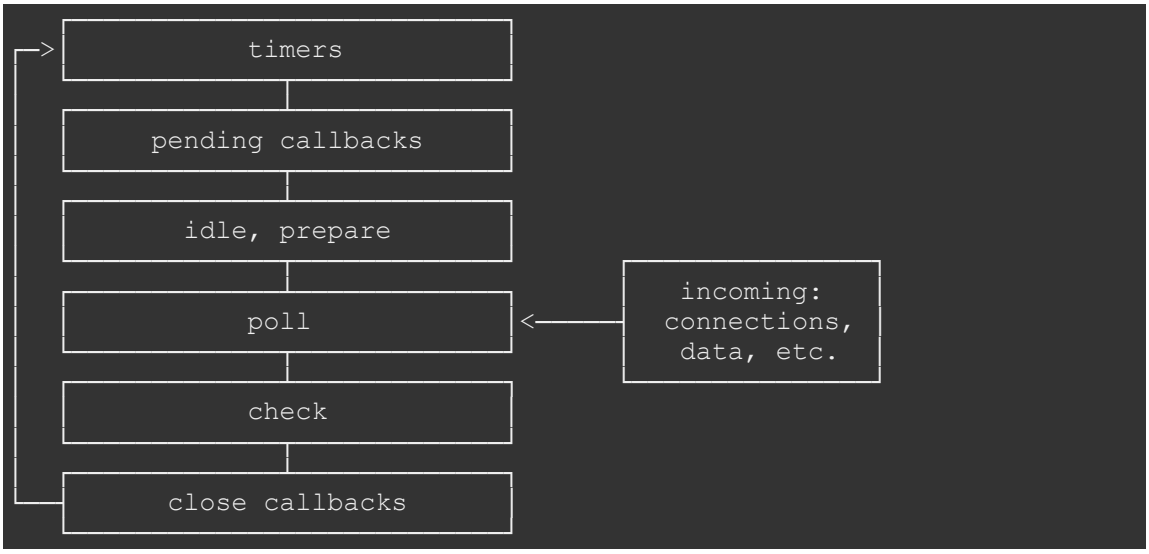
React Native on mobiili ohjelmistokehys, jonka avulla voidaan luoda *cross-platform* mobiilisovelluksia Reactia käyttäen. Tämä tarkoittaa, että mobiilisovellus

voidaan luoda JavaScriptin avulla, toisin kuin natiivisovellus, joka Androidin tapauksessa tarkoittaisi Java-pohjaista sovellusta. React Nativella kehitetään natiivi Android- ja iOS-sovelluksia. JavaScriptin käytön ansiosta mobiilikehittäminen muistuttaa paljon webkehittämistä, jolloin tekniikoita on helppo soveltaa niin web- kuin mobiilisovelluksessa. Lisäksi React Nativessa tulee mukana ominaisuus *Hot Reloading*, joka näyttää sovellukseen tehdyt muutokset reaaliajassa, toisin kuin natiivikehitysympäristöissä, joissa sovellus on käännettävä ja ajettava aina, kun halutaan tarkastella muutoksia. Yksi React Nativen suurimmista haittapuolista on kuitenkin laitteen natiivielementtien rajoitukset. Kehittäjät eivät pysty käyttämään esimerkiksi laitteen kameraa tai mikrofonia. (ReactNative)

4.2 Node.js

Sovelluksen back-end eli palvelin toteutetaan Node.js teknologialla. Node.js ajaa JavaScript koodia selaimen ulkopuolella. Tämän ansiosta palvelin voidaan rakentaa JavaScriptiä käyttäen, jolloin koko sovellus voidaan sitoa yhteen ohjelmointikieleen. (Node js)

Node.js toimii yhdessä säikeessä, eikä se luo uutta säiettä jokaisen pyynnön käsittelyyn. Node.js on suunniteltu toimimaan asynkronisesti, jonka ansiosta se pystyy käsittelemään suuren määrän pyyntöjä ja asiakkaita. Monien pyyntöjen käsittely, ilman useampaa säiettä on mahdollista Node.js *Event Loopin* avulla, joka alustetaan sovelluksen käynnistyessä. Node.js keventää palvelimen toimintaa antamalla vastuun operaatioista käyttöjärjestelmän ytimelle, eli kernelille, aina kun mahdollista. Kun kerneli on suorittanut sille annetut operaatiot, se ilmoittaa Nodelle, että tarvittavat callback-funktiot voidaan lisätä Event Loopin poll-jonoon. Event Looppiin kuuluu kuusi vaihetta. Vaikka jokaisella vaiheella on oma tarkoitus, niiden perus toiminto on aina sama. Kun vaiheeseen saavutaan, suoritetaan ensin sille ominaiset operaatiot, jonka jälkeen suoritetaan vaiheen jonossa olevat callback-funktiot. Jos kaikki pyynnöt saadaan käsiteltyä, tai vaiheen maksimi pyyntöjenkäsittelymäärä ylittyy, siirrytään seuraavaan vaiheeseen. (Node.js Event Loop) Kuviossa 8 kuvataan Event-Loopin toimintaa.



KUVIO 8 Event Loop rakenne (Node.js Event Loop)

Koska Node.js pystyy hyödyntämään vain yhtä prosessoria, se ei sovellu vaativiin palvelimiin, jotka tekevät paljon raskasta työtä. Se on suunniteltu pääosin websovelluksiin, ja suurien kyselymäärien käsittelyyn. Siksi sen skaalautuvuus on heikko.

Node.js itsessään sisältää hyvin vähän toiminnallisuuksia, mutta sen toimintaa voidaan laajentaa paketeilla. Olennainen osa Node.js pohjaisissa sovelluksissa on **npm** eli *Node package manager*. Paketteja, jotka löytyvät npm rekisteristä, voidaan asentaa helposti komentoriviltä kutsumalla *npm install <paketin nimi>*. Asennetut paketit lisätään projektin riippuvuuksiin, jotka tallennetaan *package.json* nimiseen tiedostoon. Näin ladattuja paketteja ei tarvitse lisätä esimerkiksi versionhallintaan, ja kaikki riippuvuudet voidaan asentaa komennolla *npm install*. Paketteja on lähes joka tarkoitukseen yksinkertaisista avustuskirjastoista, tehtävän ajajiin. (Node.js Introduction)

4.3 TypeScript

TypeScript on JavaScript kielen syntaktinen ylijoukko, joka käännetään JavaScriptiksi. Microsoft kehitti sen JavaScriptin ongelmien pohjalta suuria projekteja työstättäessä. TypeScriptin tarkoitus on välttää virhetilanteita, joita syntyy helposti JavaScriptiä käyttäessä. (Hejlsberg 2012) Se toi kieleen tyypityksen eli olioiden luokat. Kuviossa 9 on esimerkki JavaScriptistä.

```
function haeSalasana(salasanaTeksti) {
    if(salasanaTeksti) {
        return 'salasana';
    }
    return '********';
}

let salasana = haeSalasana('false'); // palauttaa: 'salasana'
```

KUVIO 9 Esimerkki JavaScript-syntaksista

JavaScriptissä ei voida estää *haeSalasana* funktion kutsumista väärällä, tässä tapauksessa, ei boolean arvolla, jonka takia funktio palauttaa ei halutun arvon. Tämä tilanne voidaan kuitenkin välttää TypeScript syntaksin avulla, kuten kuviossa 10 voidaan nähdä.

```
function haeSalasana(salasanaTeksti: boolean) : string {
    if(salasanaTeksti) {
        return 'salasana';
    }
    return '********';
}
let salasana = haeSalasana('false'); // Virheilmoitus
// Parametrin tulee olla boolean arvo false eikä string
```

KUVIO 10 Esimerkki TypeScript-syntaksista

Luokkien avulla voidaan varmistaa, että parametrinä tuleva arvo on varmasti oikean tyyppinen, ilman ylimääräisiä tarkistuksia funktion sisällä. Näin voidaan välttyä monilta virhetilanteilta jo ennen koodin kääntämistä. TypeScript tuo mukanaan myös muita ominaisuuksia, kuten rajapinnat, geneerisyys sekä suojamääreet. (TypeScript) Nämä ominaisuudet kannustavat hyviin ohjelmistokehitys käytäntöihin, sekä auttavat pitämään koodin helpommin luettavana ja ymmärrettävänä.

Koska TypeScript on JavaScriptin ylijoukko, se voidaan tuoda projekteihin, jotka käyttävät jo JavaScriptiä. Projekti voidaan kääntää käyttämään TypeScriptiä moduuli kerrallaan, vaikuttamatta muun ohjelman toiminnallisuuteen. Käännettävät tiedostot nimetään yksinkertaisesti *.ts* -muotoon, jonka jälkeen tehdään tarvittavat muutokset. Näin projektin kehitys voi jatkua siirtymävaiheen aikana. (TypeScript Migrating from JavaScript)

5 SOVELLUKSEN RAKENTAMINEN

5.1 Alkutoimet

5.1.1 React Native -projektin luonti ja ajaminen

Jotta React Native -projekti voitaisiin luoda, tietokoneelle täytyy asentaa Node.js, React Native command line interface sekä Android Studio. Vaikka sovellusta voi kehittää millä tahansa tekstieditorilla, Android Studio vaaditaan, koska sen mukana asennetaan tarvittavat työkalut React Native -sovelluksen rakentamiseen Androidille. Android Studion mukana asennetaan myöskin ohjelmistokehitystyökalut ja emulaattorin, jotka ovat tarvittavia testauksessa.

Kun kaikki tarvittava on asennettu, voidaan luoda uusi projekti React Native CLI:n avulla. Uusi projekti luodaan kutsumalla komentoa *react-native init <projektin nimi>*. Omassa tapauksessani projektin nimi on *client* eli asiakas. Projektissa otetaan käyttöön TypeScript käyttämällä lippua *--template typescript*.

Ohjelmaa voi testata joko fyysisellä mobiililaitteella, joka on kiinnitetty tietokoneeseen USB-kaapelilla, taikka Android emulaattorilla. Sovellus voidaan käynnistää testauslaitteessa komennolla *react-native run-android*. Kun sovellus on käännetty ja asennettu, se avautuu testauslaitteeseen.

5.1.2 Palvelimen luonti ja ajaminen

Palvelimen teknologiat ovat Node.js ja TypeScript. Uusi projekti voidaan luoda komennolla *npm init*. Komento luo tiedoston *package.json* joka sisältää tietoa Node-projektista. Tiedostoa käytetään lähinnä projektin riippuvuuksien hallitsemiseen sekä erilaisten komentojen luomiseen, joita voidaan hyödyntää esimerkiksi sovelluksen kääntämisessä, rakentamisessa ja testauksessa.

Palvelin luodaan Express kehystä käyttäen. Express ja sen TypeScript tuki voidaan asentaa komennolla *npm install --save express @types/express*. Palvelin tarvitsee myös WebSocket tuen, johon käytetään Socket.Io pakettia, joka voidaan asentaa TypeScript-tuen kanssa komennolla *npm install --save socket.io @types/socket.io*. Paketteja asentaessa voidaan käyttää *--save* lippua, joka tallentaa paketin tiedot *package.json* tiedostoon.

Pakettien asentamisen jälkeen voidaan aloittaa palvelimen rakentaminen. Palvelimen ydin on luokka *MessageServer*. Palvelin alustetaan luomalla express olio, jonka avulla voidaan kuunnella AJAX-pyyntöjä. Lisäksi täytyy luoda http-palvelin käyttämällä *createServer* funktiota. Funktiolle annetaan parametrinä juuri luotu express-olio, jolloin kaikki AJAX-pyynnöt voidaan ohjata oikeaan osoitteeseen. Kun serveri on luotu, se käynnistetään määritellyssä portissa, joka on omassa tapauksessani 8080 (kuvio 11).

```
1  import { createServer, Server } from 'http';
2  import * as express from 'express';
3
4  class MessageServer {
5
6    private app: express.Application;
7    private server: Server;
8
9    private port = 8080;
10
11   constructor() {
12     this.app = express();
13     this.server = createServer(this.app);
14
15     this.startServer();
16   }
17
18   getApp(): express.Application {
19     return this.app;
20   }
21
22   private startServer(): void {
23     this.server.listen(this.port, () => {
24       console.log(`The server is running in port ${this.port}`);
25     });
26   }
27 }
28
29 let app = new MessageServer().getApp();
30 export { app }
31
```

KUVIO 11 Palvelimen luonti ja käynnistys

Jotta luotua palvelinta voidaan käyttää, se täytyy kääntää TypeScriptistä JavaScriptiksi. Kääntämiseen tarvitaan *typescript* paketti, joka voidaan asentaa komennolla *npm install --save-dev typescript*. Typescript paketti asennetaan kehitysriippuvuutena, koska sitä ei tarvita ohjelman ajoon, vaan kehittämiseen. Kääntämisen asetukset tallennetaan tiedostoon nimeltä *tsconfig.json*. Tiedostoon asetetaan haluttu JavaScript-versio, johon sovellus käännetään sekä kansio jonne käännetyt tiedostot tallennetaan (kuvio 12).

```
1  {
2    "compilerOptions": {
3      "target": "es5",
4      "outDir": "./build"
5    }
6  }
```

KUVIO 12 tsconfig tiedosto

Kun kääntämisen asetukset on asetettu, voidaan *package.json* tiedostoon lisätä komento ohjelman kääntämiseen ja ajamiseen. Tiedoston kohtaan *scripts* voidaan luoda itsemääriteltyjä komentoja, joita voidaan ajaa komentoriviltä Noden avulla. Luotu komento nimeltä *start* kutsuu ensin komentoa *tsc* joka kääntää lähdekoodin JavaScriptiksi *tsconfig* tiedoston asetusten mukaan. Kääntämisen jälkeen ajetaan komento *node build/server.js*, joka ajaa juuri käännetyn *server.js* tiedoston, joka on kansiossa nimeltä *build*. Nyt komentoriviltä voidaan kutsua komentoa *npm start* joka kääntää ja käynnistää palvelimen (kuvio 13).

```
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1",
8     "tsc": "tsc",
9     "start": "tsc && node build/server.js"
10  },
```

KUVIO 13 package.json -tiedostoon asetettu käynnistys komento

5.2 WebSocket yhteys

WebSocket käyttöönotto on onneksi yksinkertaista Socket.IO:n ansiosta. Kaikki mitä palvelimen puolella tarvitsee tehdä, on luoda Socket.IO palvelin ja liittää se jo luotuun HTTP-palvelimeen. Kun palvelin käynnistyy Socket.IO palvelin jää kuuntelemaan, koska asiakas ottaa yhteyden. Kun asiakas yhdistetään, palvelin tiedottaa konsoliin asiasta ja tulostaa kyseisen asiakkaan WebSocket tunnisteen.

Palvelin toimii nyt lokaalissa verkossa. WebSocketin osoite on *ws://<laitteen ip4-osoite>:<portti>*. Kaikki laitteet, jotka ovat samassa verkossa voivat nyt ottaa yhteyden palvelimeen (kuvio 14).

```
1  import { createServer, Server } from 'http';
2  import * as express from 'express';
3  import * as socketio from 'socket.io';
4
5  class MessageServer {
6
7    private app: express.Application;
8    private server: Server;
9    private websocket: socketio.Server;
10
11   private address = '192.168.1.31'
12   private port = 8080;
13
14   constructor() {
15     this.app = express();
16     this.server = createServer(this.app);
17     this.websocket = socketio(this.server);
18
19     this.startServer();
20   }
21
22   getApp(): express.Application {
23     return this.app;
24   }
25
26   private startServer(): void {
27     this.server.listen(this.port, () => {
28       console.log(`The server is running in ${this.address}:${this.port}`);
29     });
30
31     this.websocket.on('connect', (socket) => {
32       console.log(`Client connected: ${socket.id}`);
33     });
34   }
35 }
36
37 let app = new MessageServer().getApp();
38 export { app }
39
```

KUVIO 14 WebSocket palvelimen luonti

Asiakkaan puolelle täytyy asentaa paketit *socket.io-client* ja *@types/socket.io-client* jotta websocket-yhteys voidaan muodostaa. Yhteyden luo singleton luokka nimeltä *Websocket*. Singleton luokkia ei voi olla ohjelmassa kuin yksi instanssi, vaikka sitä käytettäisiin monessa komponentissa. Kun yhteys luodaan singleton luokassa, voidaan varmistaa, että ohjelma muodostaa vain yhden WebSocket-yhteyden. Websocket luokan luonnin yhteydessä luodaan SocketIO yhteys palvelimelle, jonka jälkeen asiakas ja palvelin voivat aloittaa kommunikoinnin (kuvio 15).

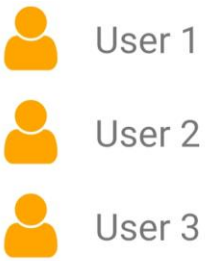
```
import SocketIOClient from 'socket.io-client';

import * as variables from '../../variables.json';
import UserService from "./UserService";

export default class Websocket {
  static get instance(): Websocket {
    if (!Websocket._instance) {
      Websocket._instance = new Websocket();
    }
    return Websocket._instance;
  }

  get socket(): SocketIOClient.Socket {
    return this._socket;
  }

  private static _instance: Websocket;

  private _socket: SocketIOClient.Socket;
  private userService = UserService.instance;

  private constructor() {
    const id = this.userService.user.deviceID;
    this._socket = SocketIOClient(`ws://${variables.server}`, {query: `user=${id}`});
  }
}
```

KUVIO 15 WebSocket yhteyden luonti asiakasohjelmassa

5.3 Käyttäjän luonti

Sovelluksen käynnistyessä asiakas lähettää get-pyynnön palvelimelle, jolla tarkistetaan, onko kyseinen käyttäjä jo tietokannassa. Käyttäjä tunnistetaan puhelimen uniikilla tunnisteella, joka saadaan *react-native-device-info* paketin avulla.

Jos kyseinen id löytyy tietokannasta, käyttäjä asetetaan *UserServicessä* tämänhetkiseksi käyttäjäksi, ja asiakas vaihtaa näkymän listaan kaikista käyttäjistä (kuvio 16), jotka löytyvät tietokannasta. Näkymä on luotu React Nativen *flatlist* komponenttia hyödyntäen. Listassa näytetään omatekoisia listaelementtejä, joissa on yksinkertainen vektori-ikoni sekä käyttäjänimi. Lista-elementtiä painettaessa, käyttäjä ohjataan keskustelu näkymään, jossa voidaan lähettää viestejä valitulle käyttäjälle. Kuviossa 17 nähdään, kuinka sovelluksen käynnistyessä lähetetään AJAX-pyyntö palvelimelle, jolla noudetaan käyttäjä laitetunnisteen avulla.



KUVIO 16 Käyttäjälista näkymä

```
private init(): void {
  const deviceID = DeviceInfo.getUniqueID();

  fetch(this.api + deviceID).then((res) => {
    if (res.status === 200) {
      res.json().then((user: User) => {
        this.initUserService(user);
        this.props.navigation.replace('UserList');
      });
    } else if (res.status === 404) {
      this.props.navigation.replace('Register');
    }
  });
}

private initUserService(user: User): void {
  UserService.instance.user = user;
}
```

KUVIO 17 Käyttäjän haku

Palvelimelle luodaan rajapinta *User*, joka laajentaa *mongoose* paketista saatavaa *Document* rajapintaa. Mongoose on Mongo-tietokannan kanssa kommunikointiin tarkoitettu ORM paketti. Dokumenttien tallentamista varten luodaan myöskin *User* niminen skeema. Vaikka Mongo on niin sanotusti *skeematon* tietokanta, ohjelmointi tasolla voidaan, ja kannattaa luoda mongoose skeema, jolloin voidaan varmistua dokumenttien datarakenteesta. Kuviossa 18 nähdään User rajapinta ja skeema.

```
import { Schema, model, Document } from "mongoose";

export interface User extends Document {
  username: string;
  deviceID: string;
}

const schema = new Schema({
  username: String,
  deviceID: String
});

export const UserModel = model<User>('UserModel', schema);
```

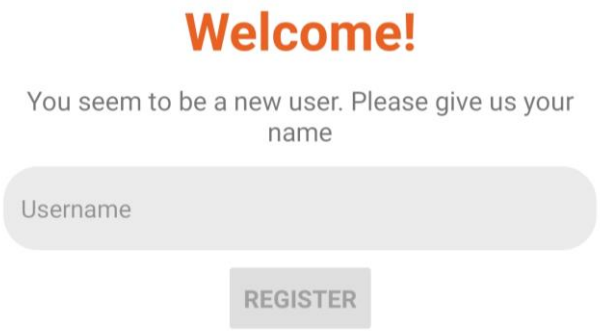
KUVIO 18 Käyttäjän datarakenne

Lähetetty kysely ohjataan *UserControlleriin*. UserController kutsuu *UserService* luokan *getUser* funktiota, joka hakee tietokannasta käyttäjää laitetunnisteen avulla. Jos käyttäjä löydetään tietokannasta UserController vastaa asiakkaalle koodilla 200, joka tarkoittaa kyselyn onnistuneen. Muutoin lähetetään 404 virheilmoitus, eli käyttäjää ei löydetty (kuvio 19).

```
router.get('/:id', async (req: Request, res: Response) => {
  const user = await userService.getUser(req.params.id);
  if (!user) {
    res.status(404).send("Could not find user");
  } else {
    res.status(200).send(user);
  }
});
```

KUVIO 19 UserController get-pyynnön käsittely laitetunnistetta hyödyntäen

Jos käyttäjää ei löydy, asiakas siirtyy rekisteröitymisnäkymään (kuvio 20), jossa voidaan syöttää käyttäjänimi. Kun käyttäjänimi on asetettu, lähetetään post-pyyntö, joka tallentaa käyttäjän tietokantaan. Post pyynnössä lähetetään asetettu käyttäjänimi sekä laitetunniste palvelimelle. UserController ottaa pyynnön vastaan ja tallentaa saadut tiedot tietokantaan UserServicen *createUser* funktion avulla (kuvio 21).



KUVIO 20 Rekisteröitymisnäkymä

```
router.post('/', async (req: Request, res: Response) => {
  const user = req.body as User;
  try {
    const created = await userService.createUser(user);
    res.status(200).send(created);
  } catch(err) {
    res.status(500).send(err);
  }
});
```

KUVIO 21 UserController post pyynnön käsittely

5.4 Viestin lähetys ja vastaanotto

Kun käyttäjälistasta valitaan käyttäjä, jolle halutaan lähettää viestejä, asiakasohjelma ohjataan keskustelunäkymään. Näkymän alustetaan lähettämällä palvelimelle get-pyyntö, joka sisältää nykyisen käyttäjän sekä keskustelukumppanin laitetunnisteet. Pyyntö ohjataan *MessageControlleriin* joka kutsuu *MessageServicen getMessagesForChat* funktiota. MessageService etsii tietokannasta tunnisteiden avulla kaikki viestit, jotka on lähetetty käyttäjien välillä, järjestää ne luontiajan mukaan vanhimmasta uusimpaan, ja palauttaa ne asiakkaalle (kuvio 22).

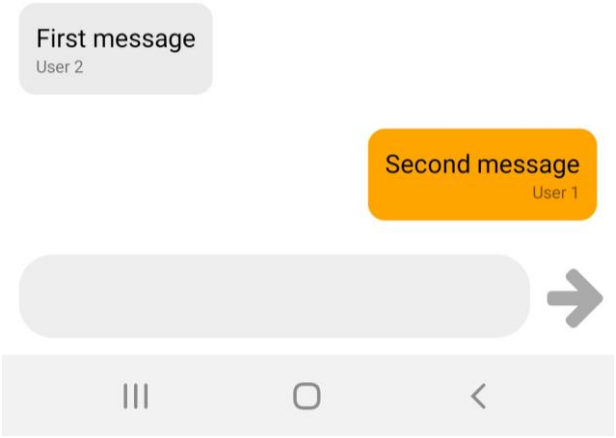
```
getMessagesForChat(sender: string, receiver: string): Promise<Message[]> {
  return Promise.all([
    MessageModel.find({'sender.deviceID': sender, 'receiver.deviceID': receiver}),
    MessageModel.find({'sender.deviceID': receiver, 'receiver.deviceID': sender})
  ])
  .then(res => {
    let messages = [...res[0], ...res[1]];
    messages = messages.sort((a, b) => {
      return a.created_ts > b.created_ts ? 1 : a.created_ts < b.created_ts ? -1 : 0;
    });
    return messages;
  });
}
```

KUVIO 22 MessageService hakee tietokannasta keskustelun viestit

Keskustelunäkymä koostuu kahdesta komponentista: *Chat* ja *ChatInput*. Chat on Flatlist-komponentti, joka listaa elementtejä käänteisessä järjestyksessä, jotta uudet viestit tulevat ruudun alareunasta. Viestit ovat myöskin omassa komponentissaan nimeltä *Message*. Viestit piirretään ruudun vasempaan reunaan harmaalla pohjalla, paitsi jos viestin lähettäjä on UserServiceen tallennettu tämänhetkinen käyttäjä, jolloin viesti piirretään oikeaan reunaan keltaisella pohjalla. Message komponentissa näytetään yksinkertaisesti viestin sisältö sekä lähettäjän käyttäjänimi (kuvio 23).

ChatInput komponentti koostuu tekstisyötteestä, sekä vektori-ikoni napista, jonka avulla voidaan lähettää kirjoitettu viesti. Aina kun tekstisyötteen sisältö muuttuu,

se talletetaan ChatInput komponentin *message* tilaan. Nappia painettaessa kutsutaan *MessageServicen sendMessage* funktiota, joka lähettää WebSocket-viestin palvelimelle. Viesti sisältää lähettäjän sekä vastaanottajan laitetunnisteen sekä viestin sisällön. Palvelin vastaanottaa viestin ja tallentaa sen tietokantaan. Tallennuksen jälkeen palvelin tarkistaa, onko viestin vastaanottaja yhdistettynä palvelimelle *connectionStoresta* (kuvio 24). Asiakkaan ottaessa WebSocket-yhteyden palvelimeen, lähetetään pyynnön mukana laitetunniste query parametrina. Kun palvelin saa tiedon yhdistämisestä, se tallentaa käyttäjän laitetunnisteen, sekä WebSocketin-tunnisteen connectionStoreen. Näin voidaan pitää kirjaa, mitkä käyttäjät ovat aktiivisena. Asiakkaan sulkiessa WebSocket-yhteyden, tiedot poistetaan connectionStoresta. Jos vastaanottaja löytyy connectionStoresta, palvelin välittää viestin vastaanottajalle WebSocket-tunnisteen avulla. Muussa tapauksessa vastaanottaja näkee viestit, kun sovellus avataan uudestaan.



KUVIO 23 Keskustelu näkymä

```
export const listenMessages = (io: Server,socket: Socket) => {
  socket.on('message', async (message: Message) => {
    const service = new MessageService();
    try {
      const mess = await service.saveMessage(message);
      const receiver = connectionStore.get(mess.receiver.deviceID);

      if (receiver) {
        io.to(receiver).emit('message', mess);
      }
      io.to(socket.id).emit('message', mess);
    } catch(error) {
      io.emit('error', error);
    }
  });
};
```

KUVIO 24 Palvelin kuuntelee uusia WebSoceket-viestejä

Asiakasohjelman *MessageService* kuuntelee jatkuvasti uusien viestien saapumista palvelimelta. Kun uusi viesti vastaanotetaan, MessageService päivittää listan keskusteluviesteistä, jotka ovat tallennettu *RxJs Observableen*. RxJs on kirjasto, joka on tarkoitettu asynkronisen ohjelmien rakentamiseen. Observableen voidaan tallentaa mikä tahansa objekti. Observablen arvoa voidaan tarkkailla, ja aina kun sen arvo muuttuu, siitä saadaan tieto. Chat komponentti tarkkailee MessageServicen *messages* observablea. Aina kun lista viesteistä päivittyy, Chat päivittää oman tilansa, jolloin komponentti piirretään uudestaan. Jos uusin viesti on käyttäjältä, jonka kanssa keskustelu ei ole juuri auki, näytetään tästä toast-ilmoitus. Kuviossa 25 on kuvattu asiakkaan MessageService luokka, joka kuuntelee saapuvia viestejä.

```
private constructor() {
  this._messages = this.messagesSubject.asObservable();
  this._message = this.messageSubject.asObservable();

  this._isLoading = this.isLoadingSubject.asObservable();

  this.updateMessages();

  Websocket.instance.socket.on('message', (data: Message) => {
    this.messageSubject.next(data);
  });
}

sendMessage(message: Message): void {
  Websocket.instance.socket.emit('message', message);
}
private updateMessages(): void {
  this.message.subscribe((data: Message) => {
    if (
      (
        data.sender.deviceID === this.currentChat &&
        data.receiver.deviceID === UserService.instance.user.deviceID
      ) ||
      (
        data.receiver.deviceID === this.currentChat &&
        data.sender.deviceID === UserService.instance.user.deviceID
      )
      ) {
      const messages = this.messagesSubject.getValue();
      messages.push(data);

      this.messagesSubject.next(messages);
    }
  });
}
```

KUVIO 25 Asiakkaan MessageService luokka kuuntelee uusia viestejä

6 POHDINTA

Tämän opinnäytetyön tarkoituksena oli kehittää full stack -sovellus, joka toimii reaaliajassa web-tekniikoin. Tuotoksena syntyi mobiilisovellus, jonka käyttöliittymä toteutettiin React Native -kirjastoa käyttäen, ja palvelin Node.js ja Express teknologioilla.

Sovelluksen reaaliaikaisuus toteutettiin WebSocket-teknologiaa käyttäen. Koska toteutettu sovellus on toiminnaltaan suhteellisen yksinkertainen, sen olisi voinut toteuttaa millä tahansa tekniikalla, joka käytiin läpi opinnäytetyössä. Erityisesti Server-Sent Events -tekniikka olisi ollut varteenotettava vaihtoehto sovelluksen toteutukseen. WebSocket on kuitenkin kaikista läpikäydyistä tekniikoista skaalautuvin. Kaksisuuntaisen kommunikoinnin ansiosta viive tiedonsiirtämisessä asiakkaan ja palvelimen välillä on pienin. WebSocket-teknologiaa soveltamalla voitaisiin toteuttaa suuriakin projekteja, jossa datan täytyy liikkua nopeasti. Olisi ollut mielenkiintoista toteuttaa sovellus myös Server-Sent Events teknologialla, sekä suurella asiakasmäärällä, jolloin oltaisiin voitu vertailla teknologioiden palvelimen kuormitusta. Näin pienellä skaalalla toteutettuna eroavaisuuksia on lähes mahdotonta huomata.

Opinnäytetyössä oltaisiin voitu toteuttaa teknisesti haastavampi sovellus, jolloin tekniikoiden eroavaisuudet olisivat tulleet paremmin esille. Kuten jo aiemmin mainittiin, sovelluksen olisi voinut toteuttaa toimivasti millä tahansa mainituista reaaliaikaisista tekniikoista. Jos opinnäytetyössä olisi toteutettu esimerkiksi pieni moninpeli, teknologioiden eroavaisuudet tulisivat paremmin esille. Tämänkaltaisessa projektissa tekniikat kuten http-polling ja long-polling karsiutuisivat pois suuren viiveen takia, ja WebSocketin kaksisuuntaisen kommunikoinnin hyödyt tulisivat esille, koska Server-Sent Eventsissä asiakkaan ainoa tapa kommunikoida palvelimelle on AJAX-pyynnöt. Työssä kuitenkin käydään läpi perusteet, kuinka WebSocket-teknologian kanssa työskennellään, ja tietoja voidaan soveltaa suurempiinkin projekteihin, joissa hyödyt tulisivat esille.

Projektista riippumatta kehittäjien kannattaa suosia push-tekniikoita lähes aina pull-tekniikoiden sijaan. Pull-tekniikat kuormittavat palvelinta turhaan, jonka takia niiden skaalautuvuus on huomattavasti heikompi. Lisäksi todellisen reaaliaikaisuuden saavuttaminen niiden avulla voi olla haastavaa, koska kyselyjä tulisi lähettää todella pienellä aikavälillä.

LÄHTEET

Dayley, B. 2014. NoSQL with MongoDB in 24 Hours. Sams Teach Yourself. Luettu 29.8.2019. https://www.informit.com/articles/article.aspx?p=2247310&seqNum=3

ESDS. 2011. Advantages And Disadvantages of Client application server. Luettu 1.6.2019. https://www.esds.co.in/blog/advantages-and-disadvantages-of-client-application-server/#sthash.3YblMC1U.61IHw11c.dpbs

Fette, I. Melnikov, A. 2011. The WebSocket Protocol. IETF. Luettu 28.8.2019. https://tools.ietf.org/html/rfc6455

Flux docs. In-Depth Overview. Luettu 29.8.2019. https://facebook.github.io/flux/docs/in-depth-overview/

Genchev, M. 2008. How do I implement basic Long Polling? Stackoverflow. Luettu 20.5.2019. https://stackoverflow.com/questions/333664/how-do-i-implement-basic-long-polling

Hejlsberg, A. 2012. What is TypeScript and why with Anders Hejlsberg. Hansel minutes. Kuunneltu 6.6.2019. https://www.hanselminutes.com/340/what-is-typescript-and-why-with-anders-hejlsberg

Hickson, I. 2015. Server-Sent Events. W3C. Luettu 28.8.2019. https://www.w3.org/TR/eventsource/#server-sent-events-intro

JavaScript.info. Server Sent Events. Luettu 28.8.2019 https://javascript.info/server-sent-events

JavaScript.info. WebSocket. Luettu 28.8.2019. https://javascript.info/websocket

Loreto, S, Saint-Andre, P & Salsano, S. 2011. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. IETF. Luettu 28.8.2019. https://tools.ietf.org/html/rfc6202

MDN web docs. Ajax. 2019. Luettu 28.8.2019. https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

MDN web docs. EventSource. 2019. Luettu 21.5.2019. https://developer.mozilla.org/en-US/docs/Web/API/EventSource

Node.js. Introduction to Node.js. Luettu 5.6.2019. https://nodejs.dev/

Node.js. The Node.js Event Loop, Timers, and process.nextTick(). Luettu 5.6.2019. https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

Oxford. Lexico. Luettu 27.8.2019. https://www.lexico.com/en/definition/full_stack

ReactJs. Luettu 29.8.2019. https://reactjs.org/

ReactJs. Introducing JSX. Luettu 29.8.2019. https://reactjs.org/docs/introducing-jsx.html

ReactJs. State and Lifecycle. Luettu 29.8.2019. https://reactjs.org/docs/state-and-lifecycle.html

ReactNative. Luettu 29.8.2019. https://facebook.github.io/react-native/

Reese, G. 2000. Database Programming with JDBC and Java, Second Edition. Oracle. Luettu 29.8.2019. https://web.archive.org/web/20110406121920/http:/java.sun.com/developer/Books/jdbc/ch07.pdf

Shin, K. & Ramanathan, P. 1994. Real-Time Computing: A New Discipline of Computer Science and Engineering. IEEE. Luettu 26.8.2019. https://rtcl.eecs.umich.edu/papers/publications/1994/ramanathan-shin-ieee-proceedings.pdf

Tutorial points. WebSockets – Events & Actions. Luettu 21.5.2019. https://www.tutorialspoint.com/websockets/websockets_events_actions.htm

TypeScript. Luettu 30.8.2019. https://www.typescriptlang.org/docs/home.html

TypeScript. Migrating from JavaScript. Luettu 30.8.2019. https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html