

KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutus

Hanne Keskisalo

SERVERLESS-ARKKITEHTUURILLA TOTEUTETTU MOBIILI-
SOVELLUS TYÖAJANSEURANTAAN

Opinnäytetyö
Helmikuu 2020



OPINNÄYTETYÖ
Helmikuu 2020
Tietojenkäsittelyn koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600 (vaihde)

Tekijä(t)
Hanne Keskisalo

Nimeke
Serverless-arkkitehtuurilla toteutettu mobiilisovellus työajanseurantaan

Toimeksiantaja
Konetyö Lihavainen

Tiivistelmä

Serverless-arkkitehtuuri valittiin opinnäytetyön aiheeksi, koska sen suosio on kasvussa ja se kehittyy jatkuvasti. Serverless kiinnosti myös henkilökohtaisesti ja opinnäytetyö oli hyvä syy oppia lisää aiheesta.

Opinnäytetyössä kehitettiin toimeksiannosta prototyyppi työajanseurantasovelluksesta. Sovelluksella voi kellottaa tehtyjä tunteja, muokata ja poistaa niitä sekä liittää tunnit projektiin ja luokitella niitä. Sovellus toimii mobiililaitteella. Sovelluksen backend toteutettiin käyttäen serverless-arkkitehtuuria ja frontend kehitettiin hyödyntäen React Native -kirjastoja. Toiminnallisen osion raportointi keskittyy siihen, kuinka frontend yhdistetään serverless-backendiin.

Sovellus toimii odotetulla tavalla, mutta tietokannan käynnistämiseen kuluu aikaa lähes 60 sekuntia, jos tietokanta ehtii sammua oltuaan määritellyn ajan käyttämättömänä. Viiveestä johtuen serverless-tietokanta ei välttämättä ole paras vaihtoehto tuotannossa olevalle sovellukselle, mutta sovelluksen kehitysvaiheessa sen käyttäminen alentaa kuluja, joihin tietokannan käynnissäoloaika vaikuttaa.

Kieli
suomi

Sivuja 51

Asiasanat
Serverless, BaaS, FaaS, AWS, React Native



THESIS
February 2020
Degree Programme in Business
Information Technology

Tikkarinne 9
80200 JOENSUU
FINLAND

Author (s)
Hanne Keskisalo

Title
A Mobile Application for Time Tracking Using Serverless Architecture

Commissioned by
Konetyö Lihavainen

Abstract

Serverless architecture was selected to be a topic of the thesis because it is gaining in popularity and technology is being constantly developed. The author also had a personal interest in the topic and writing the thesis about it was a good reason to learn more.

In the thesis, a prototype of time tracking application was developed based on the commission. The application can be used for timing of working hours, modifying and deleting them. Hours can be attached to a project and can be classified. A mobile device is needed to use the application. The backend was developed using serverless architecture and the frontend was built using React Native library. The reporting of the functional section is focused on connecting frontend to the serverless backend.

The application works as expected, but starting the database takes almost 60 seconds if the database has been unused for a specified time and has been shut down. Because of the starting delay, the serverless database might not be the best option to use in a production environment. Though, if it is used during the developing process, it might lower the costs, because it is charged based on uptime.

Language
Finnish

Pages 51

Keywords
Serverless, BaaS, FaaS, AWS, React Native

Isälle. Ikävä.

Sisältö

1	Johdanto	6
2	Konttitekнологia ja serverless-arkkitehtuuri.....	8
2.1	Konttitekнологia	8
2.2	Serverless-arkkitehtuuri	9
2.3	Kontit vs. serverless	11
2.3.1	Perustaminen.....	11
2.3.2	Kustannukset	12
2.3.3	Skaalautuvuus	13
2.3.4	Viive	14
2.3.5	Testaus	15
2.3.6	Riskit ja turvallisuus	16
3	Palveluntarjoajat serverless-arkkitehtuurissa.....	17
3.1	Amazon	18
3.1.1	Amazon Cognito ja AWS IAM.....	19
3.1.2	AWS Lambda.....	20
3.1.3	API Gateway ja App Sync.....	21
3.1.4	Amazon Aurora.....	23
3.2	Google	24
3.2.1	Google Cloud Functions	24
3.2.2	Firebase.....	25
3.3	Microsoft ja IBM.....	26
3.3.1	Microsoft Azure Functions	26
3.3.2	IBM Cloud Functions	27
4	Työkalut ja menetelmät	28
4.1	Amazon Web Services	28
4.2	React Native	29
5	Toteutus	31
5.1	Suunnittelu.....	32
5.2	Backend-palveluiden käyttöönotto.....	32
5.2.1	Tietokanta	32
5.2.2	Käyttäjien tunnistaminen.....	33
5.2.3	GraphQL-rajapinta	34
5.3	Sovelluksen frontend	37
5.3.1	Rakenne	37
5.3.2	Yhteys backendiin.....	39
6	Pohdinta	43
6.1	Kehitysprosessi, työkalut ja kustannukset	43
6.2	Jatkokehitys	45
	Lähteet	47

1 Johdanto

Opinnäytetyön tarkoituksena on luoda katsaus serverless-arkkitehtuuriin, selvittää siihen liittyviä käsitteitä ja tutustua eri palveluntarjoajiin ja heidän tuotteisiinsa. Serverless-arkkitehtuuria verrataan myös konittiteknologiaan erojen ja yhtäläisyyksien löytämiseksi. Toiminnallisen osuuden tavoitteena on kehittää Konetyö Lihavaisen toimeksiannosta prototyyppi työajanseurannan mobiilisovelluksesta. Sovelluksen backend¹ toteutetaan hyödyntäen serverless-arkkitehtuuria ja front-endiin² hyödynnetään React Native -kirjastoa, jolla saadaan rakennettua toimiva sovellus sekä Android- että iOS-käyttöjärjestelmille.

Päädyin aiheeseen, koska minun oli opinnäytetyötä varten selvitettävä, mihin ja miten mobiilisovelluksen backend voidaan laittaa pyörimään. Aikaisemmin olin tehnyt enemmän frontend-kehitystä ja backend oli jäänyt vähemmälle huomiolle. Nyt minulle tarjoutui mahdollisuus hoitaa itse kaikki alusta loppuun omassa aika-tilaussani. Halusin myös selvittää tilanteesta mahdollisimman pienillä kuluilla. Serverless tuntui aiheena mielenkiintoiselta, monipuoliselta ja omaan tilanteeseeni kokeilemisen arvoiselta. Tahdoin myös valita aiheen, josta olisi kirjoitettu vähemmän kuin esimerkiksi React Nativesta.

Kolmannen kvartaalin lopulla vuonna 2019 Google Playssa ja Apple App Storessa oli yhteensä 4,2 miljoonaa mobiilisovellusta [1]. Toisen kvartaalin aikana edellä mainituissa sovelluskaupoissa olevia sovelluksia ladattiin yhteensä 28,7 miljardia kertaa. Kun verrataan latausten määrää vuoden 2015 ensimmäisen kvartaalin vastaavaan lukuun, joka oli 15,9 miljardia, huomataan latausmäärien lähes tuplaantuneen kuluneen neljän vuoden aikana. [2.] Latausmäärien kasvamisesta selittää sekä tarjolla olevien sovellusten lisääntyminen että älylaitteiden, kuten älypuhelimien ja tablettien, yleistymisen.

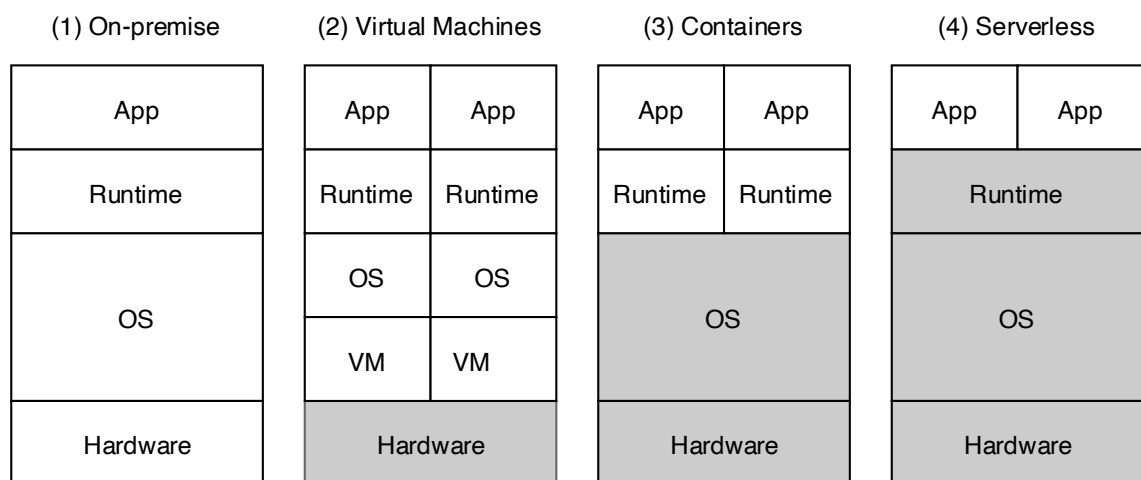
Nykyisin sovelluksia pyritään kehittämään nopealla syklillä ideasta julkaistavaksi tuotteeksi. Erityisesti startupeille on tärkeää saada tuote markkinoille mahdollisimman pian. Ensiksi ei haluta, että joku muu ehtii toteuttaa liikeidean ensin ja

¹ Backend on palvelimella katseilta piilossa.

² Frontend on se, minkä käyttäjät näkevät ruudullaan.

valloittaa markkinat. Toiseksi halutaan vähentää hävikkiä, eli hukattuja kehitysresursseja, minkä vuoksi saatetaan julkaista pienin toimiva tuote (Minimum Viable Product, MVP), jolla testataan markkinoita. MVP:n perusteella kerätään ja analysoidaan asiakaspalautetta ja kehitetään tuotetta haluttuun suuntaan. Testin perusteella halutaan myös havaita, jos tuotteelle ei ole kysyntää, jolloin projektin jatkamiseen ei käytetä turhaan enempää aikaa, vaivaa ja rahaa. [3.]

Tähän markkinarakoon, projektien testausta ja julkaisua nopeuttamaan ja kuluja pienentämään, ovat iskeneet FaaS (Function as a Service) ja BaaS (Backend as a Service) palveluntarjoajat, kuten Amazon ja Google. Muutos on-premise-palvelimista serverless-arkkitehtuuriin ei ole tapahtunut hetkessä. Kuvasta 1 nähdään, kuinka hiljalleen yhä useampi harmaalla värillä kuvattu palanen on siirtynyt omasta hallinnasta palveluntarjoajan huolehdittavaksi. Ensin vuokrattiin palvelintilaa, jossa pyöritettiin omia virtuaalikoneita [4], tämän jälkeen tuli konttitekniologia [5] ja nyt tuoreimpana tulokkaana on serverless [6].



Kuva 1. Arkkitehtuurissa on tapahtunut asteittain muutos on-premise-palvelimista serverless-toteutuksiin [mukaillen 7].

Opinnäytetyön luku 2 tutustuttaa lukijan konttitekniologian ja serverless-arkkitehtuurin ominaispiirteisiin ja niitä verrataan myös toisiinsa. Luvussa 3 esitellään muutamia tunnetuimpia pilvipalveluiden tarjoajia ja heidän tuotteitaan, jotka soveltuvat serverless-ympäristöön. Opinnäytetyön toiminnallisessa osiossa käytettyihin työkaluihin ja menetelmiin tutustutaan luvussa 4. Luvussa 5 esitellään toimeksiantajalle tehdyn sovelluksen toteutusta. Luku 6 sisältää pohdintaa ja kehitysajatuksia.

2 Konttitekнологia ja serverless-arkkitehtuuri

Luvussa esitellään sekä konttitekнологiaa että serverless-arkkitehtuuria käsitteineen. Konttitekнологia valittiin serverless-arkkitehtuurin lisäksi esittelyyn, koska sen suosio on jatkuvasti kasvussa [8].

2.1 Konttitekнологia

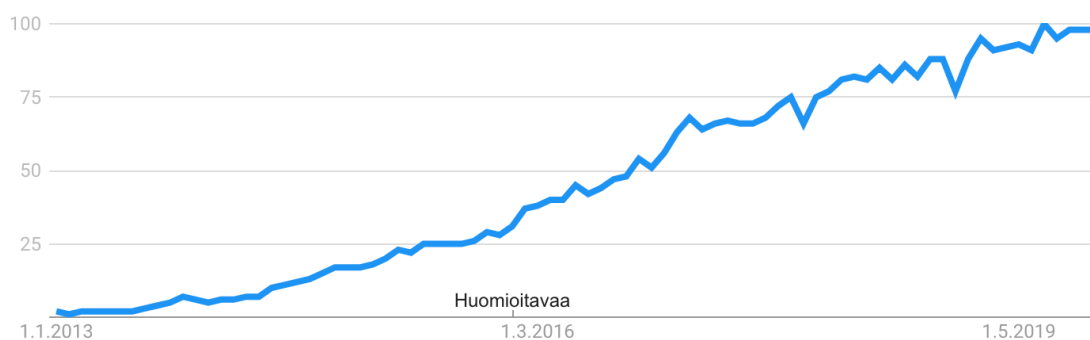
Kontti (container) on itsenäinen yksikkö, joka sisältää tarpeelliset ohjelmakirjastot, asetukset, ajettavan sovelluksen sekä riisutun version käyttöjärjestelmästä. Kontit on täysin eristetty toisistaan sekä isäntäkoneesta ja niitä ajetaan virtuaalisesti oman alustan päällä. Ensimmäinen ja edelleen suosituin alusta on Docker³. [9.]

Docker sai alkunsa vuonna 2013 avoimen lähdekoodin projektina. Toisin kuin virtuaalikoneet, jotka ovat virtualisoituja laitetasolla, kontit ovat virtualisoituja käyttöjärjestelmätasolla. Kun kontti ei ole käynnissä, se ei myöskään kuluta laitteiston resursseja. Uusia kontteja pystyy luomaan ja käynnissä olevia sammuttamaan sekunneissa. Kontti luodaan levykuvan (image) perusteella. Levykuvan sisällön voi itse määrittää ja sen pohjana voi käyttää rekisterissä olevia valmiita levykuvia. [5.]

Kuten kuva 2 osoittaa, Docker-sanan hakumäärät Google-hakukoneella ovat kasvaneet pieniä notkahduksia lukuun ottamatta. Tämä kertoo siitä, että konttitekнологia kiinnostaa kehittäjiä yhä enemmän. Tutkimusyhtiö Gartner ennustaa, että vuoteen 2022 mennessä yli 75 % yrityksistä käyttää tuotantoympäristössään kontitettuja sovelluksia [8].

Yksi konttitekнологian merkittävistä eduista on konttien siirreltävyys. Kontin voi käynnistää omalla työasemalla tai virtuaalikoneessa, Linux- tai Windows-käyttöjärjestelmällä, on-premise tai pilviympäristössä ja se toimii samalla tavalla riippumatta alustasta. [10.]

³ <https://www.docker.com/>



Kuva 2. Hakutermin ”docker” hakumäärät ajanjaksolla 1/2013–11/2019 Google-hakukoneella. Huomioitavaa-kohdassa Google paransi tietojen keräämismenetelmäänsä. [11].

IaaS (Infrastructure as a Service) tarkoittaa käytännössä pilvessä sijaitsevaa virtuaalikonetta ja mahdollisesti käyttöjärjestelmää [12]. PaaS (Platform as a Service) puolestaan sisältää infran lisäksi muun muassa kehitystä, julkaisua, testausta, hallintaa ja päivitystä helpottavia työkaluja [13]. Sekä IaaS että PaaS ovat soveltuvia alustoja konttien ajoon.

2.2 Serverless-arkkitehtuuri

Serverless, palvelimeton, ei suinkaan tarkoita sitä, että palvelimia ei enää olisi; kehittäjän ei vain tarvitse huolehtia niiden hankkimisesta, huollosta ja hallinnoinnista, kuten tavallisia palvelimia käyttämällä [14]. Palvelimista on tullut kehittäjälle näkymättömiä. Serverless-arkkitehtuurissa laiteinfra huolehtii palveluntarjoaja, jolle maksetaan käytettyjen resurssien, kuten prosessoriajan, muistin ja siirretyn datamäärän mukaan [15].

Termi serverless on ollut enemmän käytössä vuodesta 2012 lähtien, mutta AWS Lambdan ja Amazon API Gatewayn julkaisujen myötä vuosina 2014–2015 termin käyttö yleistyi [6]. Kuten kuvasta 3 nähdään, kiinnostus palvelimetonta arkkitehtuuria kohtaan on ollut nousussa vuodesta 2016 lähtien. Tämä johtunee siitä, että joulukuussa 2015 Amazonin avoimen lähdekoodin projekti JAWS (Javascript Amazon Web Services) vaihtoi nimekseen Serverless [16], mikä voi osaltaan

vauhdittaa termin yleistymistä. Nykyisin Serverless Frameworkilla on lähes 33 tuhatta tähteä Githubissa ja se tukee tunnetuimpia FaaS-funktioita [17].



Kuva 3. Hakusanan "serverless" hakumäärät ajanjaksolla 1/2015-10/2019 Google-hakukoneella. Huomioitavaa-kohdassa Google paransi tietojen keräämismenetelmänsä. [18].

Kun puhutaan serverless-arkkitehtuurista, törmätään usein käsitteisiin FaaS ja BaaS. Molemmissa tapauksissa pilvipalveluiden tarjoajan isännöimässä ympäristössä ajetaan omaa palvelinpuolen logiikkaa. Palvelua käyttävä sovellus on esimerkiksi web- tai mobiiliappi tai IoT (Internet of Things) -laite [6].

FaaS:n yhteydessä serverless tarkoittaa enemmänkin logiikkaa, funktioita, joita suoritetaan tilattomissa säiliöissä (stateless containers) jonkin ennalta määritetyn tapahtuman herättämänä. Herätin (trigger) voi olla esimerkiksi jonkin tallennetun tiedoston päivittäminen, ajastus, saapuva viesti tai rajapinnan päätepisteeseen (endpoint) saapuva HTTP-kysely. Palvelinkoodin kirjoittaminen on pitkälti samantyyppistä pilvessä kuin tavallisellakin palvelimella ja onnistuu monilla eri ohjelmointikielillä, kuten Java, JavaScript, Python ja Go. [6.] FaaS-palveluita ovat muun muassa AWS Lambda [19], Google Cloud Functions [20], Microsoft Azure Functions [21] ja IBM Cloud Functions [22].

FaaS-funktioiden voidaan ajatella olevan tilattomia (stateless), toisin sanoen ne eivät pidä luotettavasti muistissaan edellisen suorituskerran arvoja. Tilattomuudesta johtuen olisi hyvä, jos funktiot suunniteltaisiin idempotenteiksi, eli funktio antaisi samoilla arvoilla aina saman tuloksen riippumatta suorituskertojen määrästä. [23.]

FaaS-funktioilla on myös palveluntarjoajasta riippuva timeout-aika⁴, jota ei pysty kasvattamaan, mutta jonka olemassaolo on hyvä tiedostaa. Googlella aika on 9 minuuttia [24], kun taas Lambda-funktioiden timeout on 15 minuuttia [25]. Jos funktioiden suoritusaika vaihtelee runsaasti, joissain tapauksissa on mahdollista, että funktion suoritus keskeytyy. Näin voi käydä esimerkiksi tilanteessa, jossa funktion tehtävänä on pakata videotiedosto.

BaaS-palveluihin sisältyy monesti pilvessä sijaitseva tietokanta sekä autentikointipalvelu. Jos arkkitehtuurissa käytetään pelkkää BaaSia, ongelmana on se, että kaikki sovelluksen logiikka sijaitsee asiakkaan laitteella, kuten web-selaimessa tai mobiililaitteella. Yleensä osa logiikasta siirretäänkin FaaS-palvelua käyttäen pilvipalvelimelle. [6.]

Suurilla palveluntarjoajilla on olemassa palveluita sekä BaaS että FaaS tarpeisiin, ja niitä voidaan käyttää myös yhdessä. Palveluntarjoajista kerrotaan enemmän luvussa 3.

2.3 Kontit vs. serverless

Luvun tarkoituksena ei ole osoittaa serverless-arkkitehtuuria paremmaksi kuin konttitekniologiaa tai päinvastoin. Tarkoituksena on ennemminkin tarkastella kummankin teknologian piirteitä, joiden perusteella on mahdollista valita omaan käyttöön sopivampi vaihtoehto.

2.3.1 Perustaminen

Kuten luvussa 2.1 mainittiin, kontteja voi ajaa hyvin monenlaisten alustojen päällä, kuten on-premise virtuaalikoneessa tai pilvipalvelussa. Aika ja vaiva, jonka konttiympäristön pystyttäminen vaatii, riippuukin hyvin pitkälti alustasta ja sen tarjoamista mahdollisuuksista.

⁴ Ennalta määritetty aika, jonka jälkeen tapahtuman suoritus peruutetaan, mikäli määrättyä toimintoa ei ole ehditty suorittaa.

Esimerkiksi virtuaalikoneelle tarvitsee asentaa käyttöjärjestelmä sekä työkalut, kuten Docker, konttien hallintaan. Omalla vastuulla on ympäristöjen hallinta sekä päivittäminen. Jos taas käytetään PaaS-palvelua, kuten Amazonin Elastic Container Serviceä [26], kontin saa ajoon helpoimmillaan lataamalla levykuvan palveluun [27]. Ympäristön perustamisessa onkin ennakkoon tunnettava omat tarpeensa ja käytössä olevat resurssinsa, joiden perusteella valitaan sopiva paikka oman sovelluksen ajoon.

Serverless-arkkitehtuurissa ympäristön perustaminen aloitetaan valitsemalla omiin kriteereihin sopiva palveluntarjoaja. Valintakriteereinä voivat toimia esimerkiksi hinta, tarjolla olevat palvelut, käytettävissä olevat runtime-ympäristöt, integraatiotarpeet ja muiden jo olemassa olevien omien palveluiden sijainti.

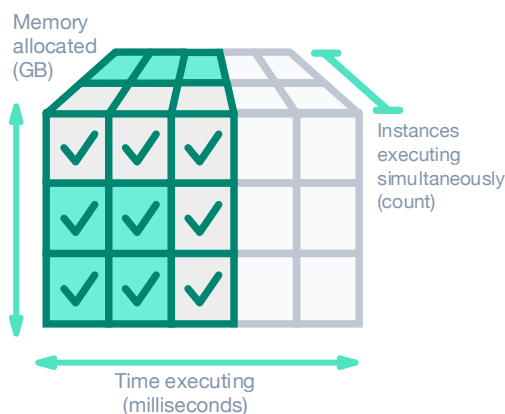
Serverless-palveluille on tyypillistä, että niiden käyttöönottoaminen on nopeaa sen jälkeen, kun on rekisteröitynyt käyttäjäksi. Esimerkiksi AWS Lambda -funktion voi luoda ja ottaa käyttöön verkkosivulla olevan konsolin kautta [28].

2.3.2 Kustannukset

Konttiteknologian käyttö- ja perustamiskustannukset riippuvat paljon siitä, mille alustalle ne laitetaan ajoon. On mahdollista ostaa ja hallinnoida omaa infraa, vuokrata palvelintilaa, vuokrata virtuaalipalvelimia tai ajaa kontteja pilvessä olevalla virtuaalikoneella tai palveluntarjoajan oman alustan päällä. Kustannuksia laskettaessa on myös huomioitava, millainen määrä henkilöstöä tarvitaan kyseisen arkkitehtuurin pyörittämiseen.

Laiteaikaa voi ostaa joko tuntiperusteisesti käytön mukaan tai maksaa palvelusta ennakkoon pidemmältä ajalta, jolloin käyttö maksaa yleensä vähemmän. Amazon tarjoaa EC2-palvelun laskutuksessa On-Demand ja Reserved Instance vaihtoehdot, joista jälkimmäinen tulee asiakkaalle jopa 60 % halvemmaksi, kun käyttöä on paljon ja sopimusaika on pitkä [29].

Serverless-arkkitehtuurissa maksetaan palveluista niiden käytön perusteella. Esimerkiksi FaaS-funktioita laskutetaan funktion suorituskertojen, niiden keston ja käytetyn laskentakapasiteetin mukaan (kuva 4) [15; 30]. BaaS-palveluilla laskutus perustuu muun muassa tehtyjen autentikointien määrään [31] tai tietokannan kokoon, käynnissäoloaikaan ja käytettyihin resursseihin [32].



Kuva 4. FaaS-palvelun hintaan vaikuttavat seikat [33].

Serverless-arkkitehtuuria käytettäessä myös koodin optimoinnin vaikutukset näkyvät suoraan kustannuksissa. Jos koodi saadaan kirjoitettua niin, että se tekee samat asiat puolet nopeammin kuin alun perin, FaaS-funktion suorittaminen maksaa 50 % vähemmän optimoituna, kun tarkastellaan vain koodin suoritukseen käytettyä aikaa.

Jos sovelluksen käyttö on jaksottaista, eli on paljon hetkiä, jolloin sovellusta ei käytetä, serverless vaihtoehto voi pienentää kustannuksia. Jos taas palvelinta kuormitetaan tasaisesti ympäri vuorokauden, esimerkiksi konttiteknologian käyttö on usein kannattavampaa ja tulee edullisemmaksi pitkällä tähtäimellä.

2.3.3 Skaalautuvuus

Kontteja on mahdollista skaalata tehtävään soveltuvan orkestrointityökalun, kuten Docker Swarmin⁵ tai Kubernetesin⁶, avulla. Kirjoitushetkellä Docker Swarm

⁵ <https://docs.docker.com/engine/swarm/>

⁶ <https://kubernetes.io/>

ei tue automaattista skaalautumista, vaan skaalaus on tehtävä manuaalisesti [34]. Kubernetes sen sijaan skaalautuu CPU:n käytön tai muun määritellyn metriikan perusteella [35]. Huomioitavaa on, että kehittäjän tulee määrittää raja-arvot, joiden perusteella konttien määrää lisätään tai vähennetään.

Yksi serverless-arkkitehtuurin ehdottomia etuja on automaattinen skaalaus. Tässä tapauksessa automaattinen tarkoittaa sitä, että palveluntarjoaja hoitaa kaiken kehittäjän puolesta. Käytännössä kehittäjä voi luottaa siihen, että palvelu, olipa kyseessä FaaS-funktio tai BaaS:n autentikointi, kykenee sopeutumaan suureen tai äkisti muuttuvaan kuormitukseen. [36.] Serverless sopii erityisesti sovellukselle, jonka tulevaa käyttäjämäärää on vaikea arvioida tai jonka käyttäjämäärä on hyvin vaihteleva.

2.3.4 Viive

Tuotannossa ajettavien konttien halutaan olevan käynnissä, kunnes toisin määrätään, jotta palvelut ovat jatkuvasti käytettävissä. Mikäli kuormitus on sopivaa, kontissa oleva tietokanta, sovellus tai palvelin toimii ilman viiveitä. [37.] Jos kontti joudutaan luomaan sen sijaan, että voitaisiin käynnistää valmis kontti, prosessi vie pidemmän ajan. Aluksi levykuva ladataan Docker Hub -rekisteristä⁷ ellei sitä ole paikallisesti saatavilla. Tämän jälkeen luodaan kontti, liitetään stdout / stderr syötevirrat terminaaliin ja luodaan verkkorajapinta. Kun nämä on tehty, kontti voidaan käynnistää. [38.]

Toisin kuin käynnissä olevan kontin tapauksessa, FaaS-funktion herätyksen ja funktion suorittamisen välissä on viivettä. Viiveen suuruuteen vaikuttaa muun muassa se, onko kyseessä kylmä- vai lämminkäynnistys ja mikä on valittu ohjelmointikieli. [39.] Viivettä lisää myös käytettävien ohjelmakirjastojen määrä [6]. Funktiokoodista kannattaakin poistaa kaikki kirjastot, joita funktio ei käytä.

⁷ <https://hub.docker.com/>

Kylmäkäynnistyksessä palveluntarjoajan alusta luo uuden kontin, johon FaaS-funktion ajonaikainen ympäristö ohjelmakirjastoineen sekä suoritettava koodi ladataan. Lämminkäynnistyksessä alusta käyttää uudelleen jo olemassa olevaan konttia, johon on jo ladattu sopiva runtime-ympäristö, jolloin funktion suoritus saadaan käyntiin nopeammin. [39.]

Eri ohjelmointikielillä on palveluntarjoajasta riippuen erilaiset ajat saada runtime-ympäristö käyttövalmiiksi. David Jacksonin ja Gary Clynchin tutkimuksen mukaan esimerkiksi AWSllä kylmäkäynnistys kestää pisimpään .NET Core 2 (2500 ms) sekä Java 8 (391 ms) runtimella ja lyhimpään Python 3.6 (2.94 ms) runtime-ympäristöllä. Lämminkäynnistyksessä nopeimmaksi osoittautui Python 3.6 (6.13 ms) ja hitaimmaksi Go (19.21 ms). Tutkimuksessa ei osannut sanoa, miksi Python runtime käynnistyi kylmänä nopeammin kuin lämpimänä. [39.]

2.3.5 Testaus

Tuotantoon vietävälle kontissa olevalle sovellukselle on mahdollista tehdä sekä yksikkö- että integraatiotestejä. Testit voi esimerkiksi suorittaa jatkuvan integraation ja julkaisun yhteydessä (CI/CD⁸) tai voidaan luoda yksi image, jossa on sekä sovellus että testausympäristö ja testidata. [40.] Konttien tapauksessa testausta helpottaa se, että kontti sisältää kaikki tarpeelliset riippuvuudet, jolloin se vastaa hyvin tuotannossa pyörivää ympäristöä [37].

Serverless-arkkitehtuurissa FaaS-funktioiden yksikkötestaus on melko suoraviivaista, koska funktiot ovat niin sanotusti tavallista koodia. Monia FaaS-funktioita voi suorittaa paikallisesti [41; 42; 43], mutta paikallinen testaus ei kuitenkaan täysin vastaa pilviympäristössä testaamista. Jos mahdolliset ongelmat eivät johdukaan omasta koodista vaan palveluntarjoajan alustasta, on enemmän kuin todennäköistä, etteivät virheet edes näy lokeissa, ellei palveluntarjoaja halua niiden näkyvän [44].

⁸ Continuous Integration / Continuous Delivery

Serverless-palveluiden yhteydessä erilaisten testattavien integraatioiden määrä on lisääntynyt. On muun muassa testattava, onko käyttöoikeuksia tarpeeksi, ovatko tietokantakyselyt oikeanlaisia ja riittääkö timeout-aika. Serverless-sovelluksen integraatiotestit pitäisi pyrkiä suorittamaan todellisten palveluiden, kuten tietokannan ja rajapinnan kanssa, jotta nähdään, miten palvelut oikeasti toimivat yhteen. [45.]

2.3.6 Riskit ja turvallisuus

Kun käytetään kolmannen osapuolen palveluita, luovutetaan suuri osa kontrollista tälle osapuolelle. Tilanteen mahdollisia riskejä ovat muun muassa yllättävät hintojen korotukset ja palveluiden ennalta-arvaamattomat alhaalla oloajat, joihin ei itse pysty vaikuttamaan ja joita ei voi itse korjata. Palvelussa voi olla rajoituksia, joista ei ole ennakoon tietoa tai uusia rajoituksia voidaan lisätä. Palvelut eivät välttämättä ole täysin kehittyneitä eikä niihin ole itse mahdollista lisätä uusia ominaisuuksia, on vain odotettava kärsivällisesti päivityksiä. [6.] Nämä riskit liittyvät yhteisesti sekä pilviympäristössä ajettaviin kontteihin että serverless-sovelluksiin.

Riskien kartoituksessa on myös hyvä miettiä, millainen prosessi on vaihtaa palveluntarjoajaa, jos se olisi tarpeellista. On tarkasteltava palveluntarjoajan luotettavuutta myös toiminnan jatkumisen näkökulmasta, eli millainen on todennäköisyys, että palvelut yllättäen poistuvat käytöstä. [46.]

Konttien siirtämisen palvelusta toiseen pitäisi olla helppoa juurikin niiden alustariippumattomuuden takia [10]. Serverless-arkkitehtuurin tapauksessa muutos FaaS-palvelusta toiseen vaatii helpoimmillaan vain pieniä koodimuutoksia, mutta isoimmillaan vaaditaan koko arkkitehtuurin muuttamista, kun osasten integrointi toisiinsa ei olekaan mahdollista tai osoittautuu liian työlääksi. [6.]

Sovellusten kanssa työskennellessä monitorointi ja debuggaus ovat tärkeässä roolissa varsinkin, kun ratkotaan ongelmia. Palveluntarjoajilla on olemassa perustasoista monitorointia, mutta tarkempia lokitietoja varsinkin lyhytikäisistä FaaS-funktioista on vaikeaa saada. [6.]

Kontteja voidaan pitää turvallisina sen vuoksi, että ne on eristetty toisistaan sekä niiden alla olevasta järjestelmästä. Turvallisuutta voidaan kuitenkin lisätä estämällä tiettyjen komponenttien pääsyn kontteihin ja rajoittamalla kontin kommunikointia tarpeettomien resurssien kanssa. [10.]

Kontteja käyttäessä kehittäjän on kiinnitettävä erityistä huomiota siihen, minkä koodin päälle rakentaa oman sovelluksensa. Vuonna 2017 tehdyn tutkimuksen mukaan Docker Hubissa olevista imageista löytyi keskimäärin 180 haavoittuvuutta. Tutkimuksessa havaittiin, että haavoittuvuuksien leviämiseen vaikuttavat vahvasti suosituimmissa pohjaimageissa (base image) olevat päivittämättömät paketit. [47.]

Serverless-arkkitehtuurin tapauksessa turvallisuutta voi parantaa huolehtimalla koko sovelluksen tietoturvasta. Estetään input-kenttien injektiot, ei anneta funktioille enempää oikeuksia kuin mitä ne tarvitsevat, käytetään monitorointia ja tapahtumien loggausta, mutta ei anneta liian tarkkoja virheraportteja loppukäyttäjälle, pidetään sovelluksen avaimet enkryptattuina ja käsitellään virheet kunnollisesti. [48.]

3 Palveluntarjoajat serverless-arkkitehtuurissa

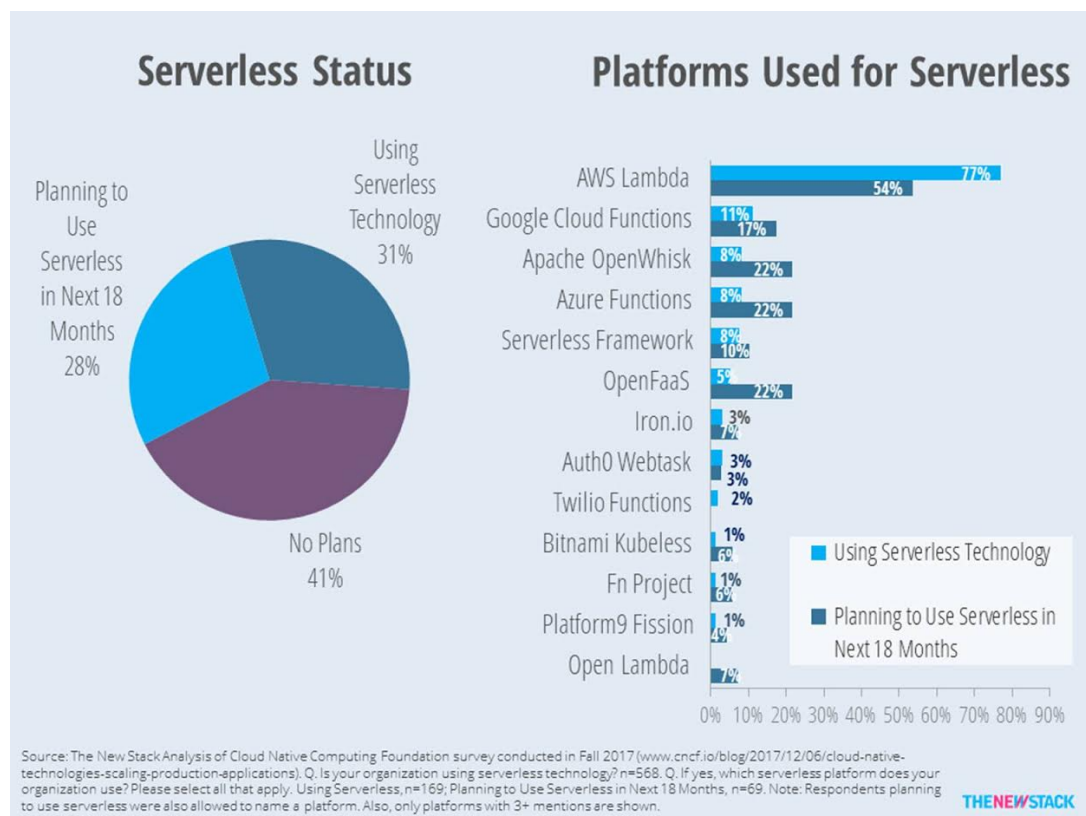
Opinnäytetyön tarkempaan tarkasteluun valikoitui neljä suurinta pilvipalveluiden tarjoajaa: Amazon, Google, Microsoft ja IBM [49]. Kuten kuvasta 5 nähdään, vuonna 2017 Yhdysvalloissa tehdyn tutkimuksen mukaan edellä mainittujen yritysten serverless-palvelut ovat nousseet käyttäjien keskuudessa suosituimmiksi. Sen lisäksi, että kyseiset yhtiöt hallitsevat tällä hetkellä yli 60 % osuutta PaaS ja IaaS markkinoista (kuva 6), niitä voi myös pitää luotettavina toimijoina, joiden palveluita pystyyneen käyttämään jatkossakin.

Opinnäytetyön ulkopuolelle jätettiin iso joukko pienempiä tarjoajia, kuten Iron Functions [50], OpenLambda [51] ja OpenFaaS [52], jotka tarjoavat vain FaaS-palvelua. Pienten tarjoajien kanssa voi myös olla isompi riski sille, että käytössä

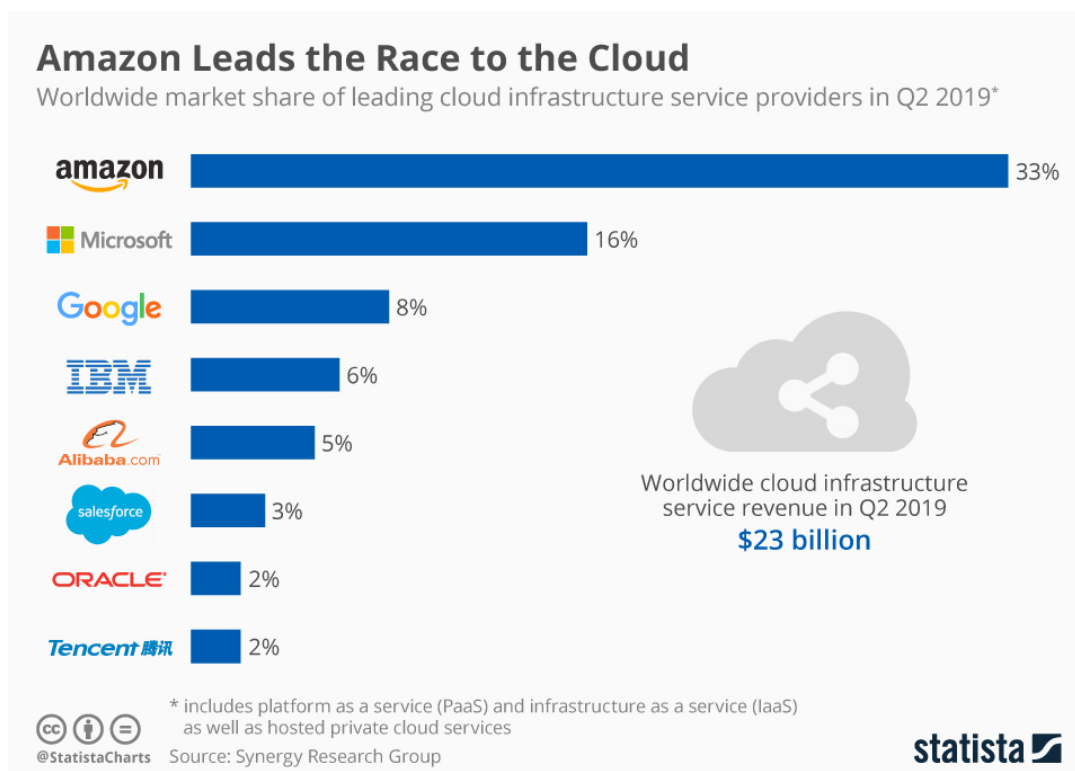
oleva palvelu lakkaa olemasta. Esimerkiksi kuvan 5 listalla oleva Auth0 Webtask [53] lopetettiin marraskuussa 2019 useamman vuoden toimintansa jälkeen [54].

3.1 Amazon

Kuten kuvista 5 ja 6 nähdään, Amazon on tällä hetkellä suurin pilvipalveluiden tarjoaja maailmassa. Amazonin pilvipalveluista käytetään yleisesti lyhennettä AWS (Amazon Web Services), ja heidän valikoimissaan on 23 kategoriaan jaettuna yli 130 tuotetta [55], joista seuraavaksi esitellään raportin kannalta oleelliset kuusi palvelua.



Kuva 5. Vuonna 2017 tehty tutkimus serverless palveluntarjoajien käyttäjämääristä [56].



Kuva 6. PaaS ja IaaS palveluntarjoajien maailmanlaajuiset markkinaosuudet vuoden 2019 toisella kvartaalilla [49].

3.1.1 Amazon Cognito ja AWS IAM

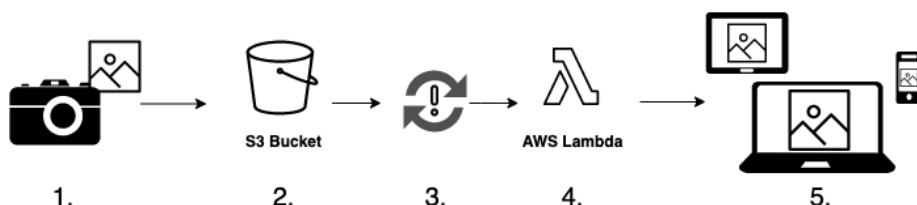
Amazon Cognito on palvelu käyttäjien rekisteröinnin, kirjautumisen ja sovellukseen pääsyn hallintaan ja se on tarkoitettu ensisijaisesti web- ja mobiilisovelluksille. Palvelu skaalautuu miljoonille käyttäjille ja tukee käyttäjänimi-salasana-kirjautumisen lisäksi kirjautumista julkisten identiteettintarjoajien, kuten Facebook, Google ja Amazon, avulla. Palvelu tukee kirjautumisessa myös SAML 2.0, OAuth 2.0 ja OpenID Connect standardeja. [31.]

Cogniton käyttäjille määritellään AWS IAM (Identity and Access Management) roolit. IAM mahdollistaa yksittäisten käyttäjien ja ryhmien roolituksen. Rooliin voidaan lisätä oikeuksia suorittaa toimintoja AWS:n sisällä tai oikeuksia voi myös vähentää [57]. Roolille voidaan esimerkiksi antaa oikeus tehdä kysely johonkin API Gatewayn päätepisteeseen tai suorittaa jokin Lambda-funktio.

3.1.2 AWS Lambda

AWS Lambda on Amazonin tarjoama FaaS-palvelu funktioiden suorittamiseen pilvessä. Lambdan tukemia runtime-ympäristöjä ovat Java, Go, PowerShell, Node.js, C#, Python ja Ruby. Näiden lisäksi on mahdollista hyödyntää Runtime APIa, jonka avulla koodin voi kirjoittaa muillakin ohjelmointikielillä. [58.]

Lambda-funktiot skaalautuvat automaattisesti, koska jokainen funktiokutsu käynnistää oman funktioinstanssin. Herätteenä voi toimia joku toinen AWS-palvelu, HTTP-päätepisteeseen tuleva kutsu tai jokin sovellus. Kuva 7 havainnollistaa Lambda-funktion toimintaa. Aluksi mobiilisovelluksella otetaan kuva, joka tallennetaan Amazonin S3 palveluun. Kuvan tallentaminen herättää Lambda-funktion, johon tallennettu koodi muuttaa kuvan koon web-selaimeen, tablettiin tai puhelimeen sopivaksi ja palauttaa kuvan laitteelle. [19.]



Kuva 7. Esimerkki Lambda-funktion herätyksestä [19].

Amazon Cognito ja AWS IAM palveluiden avulla Lambdoihin voidaan lisätä autorisointi, eli tarkistus, onko käyttäjällä oikeutta suorittaa kyseinen funktio. Näin voidaan varmistaa, että vain valtuutetut tahot voivat herättää funktion.

AWS Lambda -funktioita käytettäessä maksetaan vain kulutetusta laskenta-ajasta (compute time). Lambda-funktioiden käytöstä laskutetaan sekä käyttökerrojen että keston mukaan. Keston laskenta alkaa siitä hetkestä, kun koodin suoritus alkaa ja loppuu, kun funktio antaa palautusarvon tai koodin suoritus muuten päättyy. Kesto pyöristetään aina ylöspäin lähimpään 100 millisekunnin monikeriaan. Amazon tarjoaa joka kuukausi miljoona Lambda-funktion kutsua sekä

400 000 GB-sekuntia⁹ maksutonta käyttöä. Rajat ylittävän käytön hinnoitteluun vaikuttaa muun muassa alue, jossa palvelu sijaitsee. [15.]

3.1.3 API Gateway ja App Sync

API Gateway on vuonna 2015 julkaistu palvelu, joka mahdollistaa sovellusten vuorovaikutuksen Amazonin pilvessä sijaitsevan backendin kanssa [59]. API Gatewaylla voidaan luoda joko REST API tai Websocket API [60].

REST¹⁰-tyyppisen rajapinnan päätepisteisiin tulevat kutsut toimivat esimerkiksi Lambda-funktioiden heräteinä. HTTP-kutsun kuormana voi olla JSON¹¹- tai XML¹²-objekti. Päätepisteeseen tuleva kutsu voi myös kutsua jonkin Amazonin ulkopuolella sijaitsevaa julkista HTTP-päätepistettä. [60.]

Koska API Gateway tukee myös websocketteja¹³, sitä voidaan hyödyntää sovelluksissa, jotka tarvitsevat reaaliaikaista kaksisuuntaista datan siirtoa. Tällaisia sovelluksia ovat esimerkiksi chatit ja streamit. [60.]

API Gateway tarjoaa käyttäjän autentikoinnin hyödyntäen AWS Cognito ja AWS IAM palveluita. API Gatewayn avulla on myös mahdollista jaella useampaa eri versiota omasta APIsta samaan aikaan. Monitorointiin voi käyttää Amazonin CloudWatchia, joka tallentaa muun muassa dataa API-kutsuista, viiveestä ja virheistä. [60.]

Toinen mahdollisuus rajapinnan luomiseen on AppSync, joka julkistettiin vuoden 2017 lopussa vastaamaan erityisesti mobiilikehittäjien tarpeisiin. Toisin kuin API Gateway, AppSync käyttää GraphQL-viestintäprotokollaa. [61.] Kaikki GraphQL-

⁹ Keston yksikkö saadaan, kun kerrotaan laskentaan käytetyn muistin määrä CPU-ajalla. CPU-aika pienenee, kun laskentaan käytetään enemmän muistia.

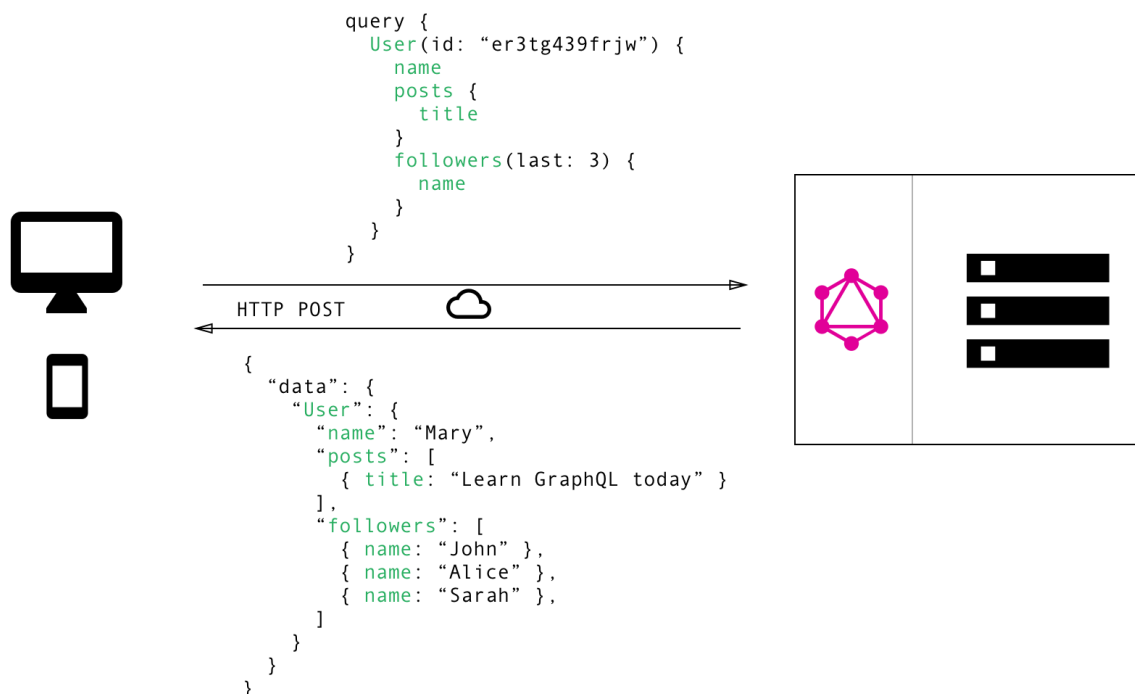
¹⁰ Representational State Transfer on arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen. Yksittäisillä resursseilla on uniikit päätepisteet, kuten `/users/{id}`.

¹¹ JavaScript Object Notation on standardoitu tiedostomuoto tiedon lähetykseen.

¹² Extensible Markup Language on laajennettava merkkaukieli rakenteisen tiedon tallentamiseen ja lähetykseen.

¹³ Websocket on standardoitu kaksisuuntainen reaaliaikainen selaimen ja palvelimen välillä oleva yhteys.

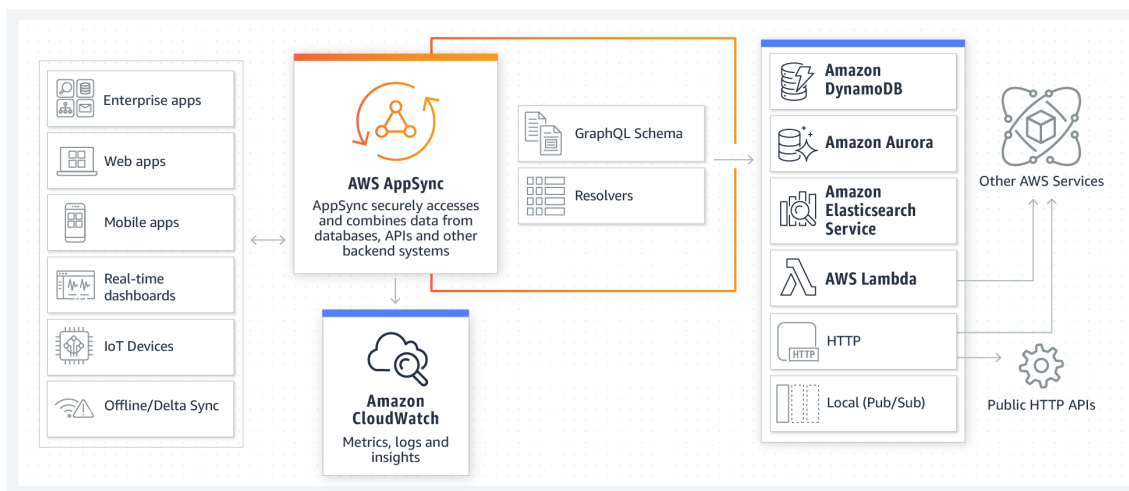
kyselyt sisältävät määrittelyn siitä, millaista palvelimelta haettava data on rakenteeltaan ja näin ollen vastaus palauttaa vain tarpeelliset tiedot. GraphQL-kyselyn rakennetta havainnollistaa kuva 8. Verrattuna REST-rajapintaan, joka on resursipohjainen ja palauttaa tiettyyn päätepisteeseen tehtyyn kyselyyn aina tietynrakenteisen vastauksen, GraphQL:n käyttö voi vähentää tehtävien kyselyiden määrää. [62.]



Kuva 8. Esimerkki GraphQL-kyselystä, jossa määritellään, mitä tietoja ja missä muodossa ne halutaan saada palvelimelta [62].

API Gatewaysta poiketen AppSync tarjoaa myös offline-tuen, jonka seurauksena käyttäjän on mahdollista vuorovaikuttaa datan kanssa ja päivittää dataa myös silloin, kun nettiä ei ole saatavilla. Kun nettiyhteys palautuu, AppSync synkronoi muuttuneet tiedot tietokantaan. [62]

Kuvasta 9 nähdään, että AppSync-palvelua on mahdollista käyttää hyvinkin erilaisiin tarpeisiin, kuten mobiili- ja web-sovelluksiin sekä IoT (Internet of Things)-laitteisiin. Monitorointi voidaan hoitaa Amazon CloudWatch -palvelulla ja dataa voidaan hakea niin NoSQL- kuin relaatiotietokannoista [63]. Käyttäjien tunnistus voidaan tehdä AWS IAMn ja Amazon Cogniton avulla, kuten API Gatewaynkin tapauksessa [62].



Kuva 9. Amazon AppSync -palvelu integroituu niin osaksi web- ja mobiili-sovelluksia kuin IoT-laitteitakin. Data on mahdollista tallentaa joko NoSQL- tai SQL-tietokantaan. [64.]

3.1.4 Amazon Aurora

Amazon Aurora on MySQL- ja PostgreSQL-yhteensopiva pilvessä sijaitseva reaaliaikainen tietokanta. Perinteisten aina käynnissä olevien vaihtoehtojen lisäksi Amazon tarjoaa serverless-tietokannan, joka automaattisesti käynnistyy, sammuu ja skaalautuu asiakassovelluksen tarpeiden mukaan. Serverless-vaihtoehto on sovelloppainen silloin, kun tietokannan käyttö on ajoittaista ja ennustamatonta ja kun käyttäjämäärät vaihtelevat. [65.]

Amazon Aurora Serverless -tietokannan käytöstä laskutetaan sekuntiperusteisesti siitä ajasta, kun tietokanta on käynnissä, koska tietokannalla ei ole ennakoon määritettyjä aina käynnissä olevia instansseja. Laskua kertyy tietokannan koon, I/O-liikenteen ja tietokannan kapasiteetin mukaan. Tietokannan kapasiteetilla on oma yksikkö, ACU (Aurora Capacity Unit). Yhdellä ACUlla on käytössään keskimäärin 2 GB muistia sekä prosessoritehoa. Serverless-tietokannasta ei ole kirjoitushetkellä saatavissa maksutonta Free Tier -kokeilua vaan kaikki käyttö on maksullista. [32.]

Serverless-tietokannalle ei ole mahdollista määrittää julkista IP-osoitetta, mutta siihen pääsee käsiksi joko Lambda-funktioiden avulla tai Data API -palvelua käyttämällä. Käytännön on tarkoitus lisätä palvelun tietoturvaa estämällä luvattomista lähteistä saapuvat kyselyt. [66.]

3.2 Google

Vuonna 2008, kaksi vuotta Amazonia myöhemmin, Google julkaisi ensimmäisen pilvessä toimivan palvelunsa, App Enginen, esikatseluversion [67]. Nyt 11 vuotta myöhemmin Googlessa on pilvipalveluita jo yli 100 ja erityisesti palvelut tekoälyyn ja koneoppimiseen ovat valikoimassa hyvin edustettuina [68].

3.2.1 Google Cloud Functions

Google Cloud Functions on Googlen tarjoama FaaS-palvelu, jonka ensimmäinen versio julkaistiin alkuvuodesta 2017 [69]. Funktioita voi suorittaa Node.js, Python 3 ja Go runtime-ympäristöissä. Funktion suoritus käynnistyy, kun ennalta määritetty tilanne, kuten tiedoston lataus pilvitallennustilaan, muutos lokissa tai saapuva viesti, tapahtuu. [70.] Google tarjoaa myös oman HTTP-herätteen, jonka avulla funktiota varten määritettyyn HTTP-päätepiisteeseen tuleva kysely herättää nimetyn funktion. Esimerkiksi komentorivillä tehtyä POST-tyyppistä kyselyä

```
curl -X POST  
"https://YOUR_REGION-YOUR_PROJECT_ID.cloudfunctions.net/FUNCTION_NAME"  
-H "Content-Type:application/json" --data '{"name":"Keyboard Cat"}'
```

voidaan käyttää herätteenä [71].

Palvelusta laskutetaan käytön mukaan, ja se skaalautuu automaattisesti. Palvelun käyttöönotossa voidaan määrittää, kuinka tehokas suoritin ja minkä verran muistia funktioiden suorittamiseen tarvitaan. [70.] Funktioiden skaalautumista on myös mahdollista rajoittaa, jos esimerkiksi tietokanta ei pysty käsittelemään suurta määrää yksittäisiä kutsuja [24].

Google tarjoaa kuukausittain 2 miljoonaa funktiokutsua, 400 000 GB-sekuntia ja 200 000 GHz-sekuntia laskenta-aikaa sekä 5 GB verkkoliikennettä maksutta. Verkkoliikenne Googlen omien APIen sisällä on aina maksutonta. Laskenta-aika pyöristetään ylöspäin lähimpään 100 millisekunnin monikertaan, eli 260 millisekunnin käytöstä joutuu maksamaan 300 millisekunnin mukaan. [72.]

Funktioiden turvallisuutta voidaan lisätä käyttämällä hyväksi Google Service Accountiin perustuvaa autentikointia [70]. Funktioita on mahdollista testata paikallisessa ympäristössä käyttämällä Firebase CLIn mukana tulevaa Cloud Functions emulaattoria, joka matkii paikallisesti pilviympäristöä [41].

3.2.2 Firebase

Firebase on BaaS-palvelu, jonka Google osti vuonna 2014. Palvelun on tarkoitus nopeuttaa ja helpottaa web- ja mobiilisovellusten kehittämistä ja julkaisua. [73.]

Käyttäjiä voidaan autentikoida joko käyttämällä ulkoista salasanaa, puhelinnumeroa tai federoitua identiteetitarjoajaa, kuten Googlea, Facebookia tai Twitteriä. Firebase tukee sekä OAuth 2.0 että OpenID Connect standardeja. [74.]

Firebaseen voi yhdistää monipuolisesti erilaisia ominaisuuksia. Firebase tukee Cloud Functions -funktioiden käyttöä, jolloin osan sovelluksen toiminnallisuudesta voi siirtää funktioiden hoidettavaksi. Dataa voi tallentaa Cloud Firestoreen, joka on pilvessä sijaitsevaa NoSQL tietokanta. Tiedostojen tallennuspaikkana toimii Cloud Storage. Realtime Database sopii sovelluksiin, joissa on tärkeää datan reaaliaikainen tallentaminen. Google tarjoaa myös kattavat työkalut muun muassa analytiikkaan ja viestien lähettämiseen laitteille. [75.]

Yksi Cloud Firestore -tietokannan parhaita ominaisuuksia on se, että käyttäjällä on pääsy tietokantaan, vaikka laite olisi offline-tilassa. Cloud Firestore tekee tietokannasta laitteen välimuistiin kopion, johon voi tehdä kyselyitä ja tallettaa uutta dataa. Kun laite saa jälleen yhteyden verkkoon, tietokannat synkronoituvat automaattisesti. [76.]

Vuonna 2018 Google esitteli ML Kit Betan, joka on SDK¹⁴ koneoppimiseen. ML Kit sisältää valmiina viisi APIa, joilla pääsee alkuun muun muassa tekstin, kasvojen ja maamerkkien tunnistamisessa, viivakoodien lukemisessa sekä kuvien otsikoinnissa. [77.]

3.3 Microsoft ja IBM

Microsoft ja IBM ovat viimeiset raportissa esiteltävät palveluntarjoajat. Molemmat luottavat yhteisön voimaan kehittäessään omia FaaS-palveluitaan.

3.3.1 Microsoft Azure Functions

Microsoft julkaisi Azure Functions -funktiot yleiseen käyttöön vuoden 2016 lopulla [78]. Funktioita kehitettiin ennen julkaisua yhteistyössä kehittäjäyhteisön kanssa ja projektilla on edelleen oma repo GitHubissa [79].

Azure Functions CLI mahdollistaa funktioiden paikallisen testaamisen. Työkalu voidaan asentaa sekä Windows-, Mac- että Linux-käyttöjärjestelmään. Funktioita voi kehittää ja testata joko paikallisesti tai Azuren omassa portaalissa, mutta ei molemmissa. [42.] Funktiot on mahdollista yhdistää muihin Azuren palveluihin, kuten Blob Storageen, Event Hubiin, Service Busiin, sekä ulkoisiin palveluihin, kuten OneDriveen ja DropBoxiin [78].

Azure Functions -palvelua voi käyttää kuukausittain maksutta 400 000 GB-sekunnin verran ja tehdä miljoona funktiokutsua. Ylimenevästä käytöstä laskutetaan funktioiden suorittamiseen käytetyn ajan mukaan. Aika pyöristetään ylöspäin lähimpään 100 millisekunnin monikertaan. [78.]

Azurella on olemassa kaksi eri runtime-versiota funktioiden suorittamiseen ja uusi tuleva versio 3.x on esikatselussa. Valittava runtime-versio riippuu vastaavasta

¹⁴ SDK (Software Development Kit) on tietylle ohjelmointikielelle tuotettu paketti ohjelmointia helpottavia työkaluja, jotka voi ladata kerralla pakettienhallintajärjestelmän, kuten npm, yarn tai pip, kautta.

.NET versiosta. Runtime 2.x ympäristössä on tuki C#, Java, F#, JavaScript, Python ja PowerShell ohjelmointikielille. Kun ohjelmointikieli on valittu, kaikkien funktiosovelluksessa (function app) olevien funktioiden tulee olla kirjoitettu samalla kielellä. [80.]

3.3.2 IBM Cloud Functions

IBM Cloud Functions -palvelu on jalostettu käyttämällä avointa Apache OpenWhisk lähdekoodia. IBM Research toimi OpenWhisk-projektin ensimmäisenä kehitysryhmänä ennen kuin Apache otti sen mukaan kehitysohjelmaansa vuonna 2016. [81.]

IBM tarjoaa Node.js, Python, Swift, PHP, Go, Ruby, Java ja .NET core runtime-ympäristöt funktioiden suorittamiseksi eri ohjelmointikielillä. IBM mahdollistaa myös minkä hyvänsä muun ohjelmointikielen käyttämisen Docker-konttien avulla, joten kukin kehittäjä voi käyttää haluamaansa kieltä funktioiden luomiseen. [82.]

Funktioiden maksimisuoritus aika voidaan asettaa 10 minuuttiin ja tarjolla on enintään 2048 MB muistia [82]. Funktioita voi suorittaa 400 000 GB-sekunnin edestä maksutta kuukausittain. Määrän ylittävästä käytöstä laskutetaan, kuten muillakin palveluntarjoajilla, ajan ja muistin mukaan lähimpään 100 millisekunnin moniker- taan ylöspäin pyöristettynä. [33.]

Kaikki funktioiden lokitiedot tallentuvat automaattisesti ja niiden yhteenvetoja suorituskyvystä ja terveydestä pystyy tarkastelemaan graafisesti IBM Cloud Monitoring -palvelun avulla. Monitorointi hyödyntää Grafanaa¹⁵ ja kehittäjän on esimerkiksi mahdollista asettaa sähköpostihälytyksiä raja-arvojen ylittyessä. [83.] Funktioiden suorituksen lokeja on mahdollista analysoida käyttämällä IBM:n LogDNA-palvelua, joka indeksoi lokitapahtumat, mahdollistaa sanahaun ja kyse- lyt eri kenttien tiedoista [84].

¹⁵ Grafana on metriikan analysointi- ja visualisointityökalu. <https://grafana.com/>

Serverless-backendin rakentamiseksi IBM tarjoaa Cloud Functions -funktioiden lisäksi API Gateway REST-rajapinnan, Cloudant JSON-dokumenttitietokannan [85] sekä App ID autentikointipalvelun, joka on tarkoitettu erityisesti web- ja mobiilisovelluksille [86]. Myös tekoälyalusta Watson on mahdollista konfiguroida osaksi arkkitehtuuria [87].

4 Työkalut ja menetelmät

Luvussa käydään läpi, mitä AWS:n palveluita on valikoitu käytettäväksi opinnäytetyön toiminnallisessa osiossa ja mistä syistä. Lukijalle kerrotaan myös, mikä React Native on ja miten sitä hyödynnetään frontend-kehityksessä.

4.1 Amazon Web Services

Työajanhallintasovelluksen backend päätettiin toteuttaa käyttäen AWS:n palveluita. Kuten luvussa 3.1 mainittiin, Amazon on maailman johtava FaaS ja BaaS palveluiden tarjoaja. Tämä tarkoittaa myös sitä, että työelämässä tulee olemaan hyötyä Amazonin palveluiden tuntemisesta ja käyttökokemuksesta, jota opinnäytetyön toteutusvaiheessa kertyy. Google oli varteenotettava kilpaileva vaihtoehto Amazonille, mutta Firebase ei tue relaatiotietokantaa, minkä vuoksi palvelua päätettiin olla käyttämättä.

Käyttäjän autentikointi toteutetaan **Amazon Cognito** avulla, koska samalla kirjautumisella saadaan hoidettua niin sovellukseen kuin rajapintaankin tunnistautuminen. Kirjautuessa käyttäjä saa pääsyyn oikeuttavan käyttöoikeustietueen (access token) ja päivitykseen käytettävän käyttöoikeustietueen (refresh token). Access token on voimassa rajoitetun ajan ja se voidaan uusida refresh tokenin avulla ilman, että käyttäjän on kirjauduttava uudelleen sovellukseen.

Cognitossa access token on voimassa yhden tunnin [88] ja se on liitettävä mukaan kaikkiin rajapinnalle tehtäviin kyselyihin osoittamaan, että käyttäjällä on

pääsy pyydettyyn resurssiin. Refresh tokenin voimassaoloaika voidaan määrittää välille 1–3650 vuorokautta [88].

Sovelluksen tietokantana on **Amazon Aurora Serverless MySQL**, joka on reaaliaikainen tietokanta. Serverless-tietokannalle voidaan ottaa käyttöön Data API -ominaisuus, jonka avulla tietokantaan voi tehdä kyselyitä joko selaimen AWS-hallintapaneelista löytyvän Query Editorin tai AppSyncin kautta. Tietokantaan kirjaututaan joko käyttäjätunnus-salasana-yhdistelmällä tai turvallisemmin Secrets Manageria [89] hyödyntämällä. Käyttäjätunnus ja salasana voidaan tallentaa AWS:n Secrets Manageriin, jolloin riittää, että käytetään Managerista saatua ARN-tunnistetta (Amazon Resource Name). Palvelulle, esimerkiksi AppSync, on erikseen myönnettävä pääsy kyseiseen tunnisteeseen, joten kuka tai mikä hyvänsä ei voi käyttää tunnistetta.

Sovelluksen rajapintavaihtoehtoja olivat API Gateway ja AppSync, joista valittiin jälkimmäinen sen vuoksi, että siinä pystyy myöhemmin ottamaan käyttöön offline-ominaisuuden. AppSyncin etu oli myös se, että GraphQL-rajapintaa käyttämällä vähennetään rajapinnalle tehtävien kyselyiden määrää ja niin sanotusti turhaa datasiirtoa.

4.2 React Native

React Native on Facebookin kehittämä JavaScript-kirjasto mobiilikäyttöliittymien rakentamiseen. Kirjasto julkaistiin vuonna 2015 ja sitä on kehitetty julkisesti siitä lähtien [90]. Vuonna 2018 React Nativella oli toiseksi eniten kehittäjiä, kun vertailtiin kaikkia Githubin repositoreja [91], mikä kertoo omaa tarinaansa kirjaston suosiosta.

Natiivisti Android-sovellukset koodataan Javalla ja iOS-sovellukset Swiftillä. React Native mahdollistaa koodin kirjoittamisen JavaScriptillä sekä Androidille että iOS:lle. Tämä helpottaa koodin ylläpidettävyyttä, kun eri käyttöliittymille ei tarvita omia tiimejä. [90.]

React Native on pitkälti samanlainen kuin webkäyttöliittymien rakentamiseen tarkoitettu kirjasto React, mutta web-komponenttien sijaan käytetään käyttöliittymän natiiveja komponentteja [90]. Komponentit kirjoitetaan yleensä JSX-syntaksilla (JavaScript XML), joka on samankaltainen kuin HTML-syntaksi. Esimerkiksi kaikki tagit tulee sulkea, joko `<komponentti>...</komponentti>` tai `<komponentti/>`. JSX:ssä JavaScriptiä on mahdollista kirjoittaa aaltosulkujen sisälle, jolloin voidaan luoda dynaamista sisältöä. [92.]

Reactista puhuttaessa on hyvä ymmärtää kaksi tärkeää käsitettä, propsit (prop, properties) ja tila (state). Propsit välittävät tietoa eri komponenttien välillä ja ne lisäävät komponenttien uudelleenkäytettävyyttä. Parent-komponentti asettaa propsit ja ne säilyvät muuttumattomina komponentin elinajan. Esimerkiksi parent-komponentti

```
const person = <Person occupation="Software Developer"/>;
```

antaa propsina ammatin. Person-luokassa määritellään, kuinka parentilta saatua propsia käytetään

```
class Person extends React.Component {
  render() {
    return <View>I am a {this.props.occupation}!</View>;
  }
}
```

Tässä näytölle tulostuu teksti I am a Software Developer.

Jos dataa on tarve muuttaa, käytetään komponentin tilaa, joka on alustettu komponentin konstruktorissa. Tilan muutos tapahtuu monesti käyttäjän toiminnan, kuten tekstin kirjoittamisen, tai palvelimelta ladatun datan seurauksena. Tilan muuttuminen aiheuttaa komponentin uudelleen renderöinnin, jolloin muuttunut data päivitetään näkyviin. Tässä React Nativen esimerkissä [93] tila muuttuu ajastimella sekunnin välein ja teksti I love to blink vilkkuu.

```
class Blink extends React.Component {
  constructor(props) {
    super(props);
    /** state object */
```

```

    this.state = { isShowingText: true };
  }

  componentDidMount(){
    /** Toggle the state every second */
    setInterval(() => (
      this.setState(previousState => (
        { isShowingText: !previousState.isShowingText }
      ))
    ), 1000);
  }

  render() {
    if (!this.state.isShowingText) {
      return null;
    }
    return <Text>{this.props.text}</Text>;
  }
}

export default class BlinkApp extends React.Component {
  render() {
    return (
      <View>
        <Blink text='I love to blink' />
      </View>
    );
  }
}

```

Tila on komponenttikohtainen, eikä siihen pääse käsiksi toisesta komponentista. Jos tilatieto on kuitenkin tarpeellista muiden komponenttien toiminnalle, on mahdollista käyttää tilasäiliötä (state container), kuten Redux [94] tai MobX [95], joista voi lukea lisää viitteistä.

5 Toteutus

Sovelluskehitys lähtee käyntiin huolellisella suunnittelulla. Kun suunta on selvillä alkaa varsinainen kehitystyö. Luvussa kerrotaan, miten eri palveluita otettiin käyttöön ja kuinka frontend saatiin toimimaan yhteistyössä backendin kanssa.

5.1 Suunnittelu

Sovelluksen kehitys lähti käyntiin käyttötapauksien kartoittamisella. Toimeksiantajan kanssa mietimme, mitä valmiilla sovelluksella haluttiin pystyä tekemään ja mitä asioita sen oli tarkoitus helpottaa. Päädyimme seuraaviin kokonaisuuksiin:

1. Työtuntien kellottaminen ja luokittelu sekä työlajeittain että projekteittain.
2. Suodatettujen raporttien tuottaminen. Suodatin voi olla esimerkiksi aikaväli tai projekti.
3. Ajopäiväkirja paikannusta hyödyntäen.

Opinnäytetyö rajattiin koskemaan vain listan ensimmäistä kokonaisuutta, eli tuntien kellotusta ja luokittelua, jotta käytettävissä oleva aika riitti toimivan vaikkakin toiminnallisuuksiltaan rajatun sovelluksen toteuttamiseen.

5.2 Backend-palveluiden käyttöönotto

Ennen kuin frontendiä päästiin kunnolla kehittämään, oli otettava käyttöön tarvittavat backend-palvelut. Luvussa kerrotaan, missä järjestyksessä palvelut pystytettiin ja millaisia asetuksia on käytetty. Kaikki palvelut on luotu Irlantiin, eli alueelle EU-WEST-1, koska opinnäytetyön kirjoitushetkellä Aurora Serverless -tietokantaan liittyvät uudet ominaisuudet, kuten Data API, tulivat kaikista Euroopan palvelimista ensimmäisenä Irlannin palvelimelle.

5.2.1 Tietokanta

Projektia varten luotiin Amazon Aurora Serverless MySQL-relaatiotietokanta, jonka Data API -ominaisuus laitettiin päälle. Data APIa käyttämällä tietokannalla on yksi pääteosoite, johon kaikki kyselyt tehdään. Tietokannalle asetettiin 15 minuutin pysäytysaika. Tietokanta siis odottaa viimeisimmän kyselyn jälkeen määritetyn ajan, jonka jälkeen kapasiteettiyksiköiden määrä lasketaan nolnaan, jolloin tietokanta niin sanotusti nukkuu eikä kuluta resursseja ja aiheuta kuluja.

Tietokannan taulut luotiin Query Editoria käyttämällä. Editorissa voi käyttää tavallisia MySQL-komentoja ja käytettävän tietokannan nimi on kerrottava jokaisen komennon yhteydessä. Tietokantaan luotiin uusi käyttäjä, jolle annettiin CRUD¹⁶-oikeudet tietokannan tauluihin. Uuden käyttäjän käyttäjänimi ja salasana tallennettiin Secrets Manageriin ja ARN-tunniste kirjoitettiin muistiin.

5.2.2 Käyttäjien tunnistaminen

Amazon Cognitoon luotiin uusi User Pool, johon käyttäjien tiedot, kuten käyttäjänimi ja salasana, tallennetaan. Asetuksista valittiin, että sähköpostiosoitetta voi käyttää kirjautumisessa käyttäjänimenä, jolloin Cognito pitää huolen siitä, että sähköpostiosoitteiden tulee olla uniikkeja. Määriteltiin, että salasanassa pitää olla vähintään 10 merkkiä ja sen tulee sisältää isoja ja pieniä kirjaimia sekä numeroita. Cognitoissa olisi mahdollista ottaa käyttöön monimenetelmäinen todentaminen (Multi-Factor Authentication, MFA), jolloin jokaisen kirjautumisen yhteydessä pitäisi syöttää esimerkiksi sähköpostiin saapunut kertakäyttöinen koodi, mutta sitä ei nähty tarpeelliseksi ottaa käyttöön.

Rekisteröitymisen yhteydessä annettava sähköpostiosoite on vahvistettava ennen kuin rekisteröinti on loppuun saakka suoritettu. Koodin sisältävä sähköpostiviesti on mahdollista lähettää automaattisesti Cogniton kautta, mutta on pidettävä mielessä, että yhdestä User Poolista voi lähettää päivässä maksimissaan 50 viestiä [88]. Jos tarve olisi suurempi, niin Amazon suosittelee ottamaan käyttöön Simple Email Servicen (SES) [96].

Jotta sovellus saa yhteyden Cognitoon luotuun User Pooliin, tulee sinne luoda myös App Client. On tärkeää, että clientin luonnin yhteydessä poistetaan raksi ”Generate Client Secret”, koska sitä ei tarvita JavaScript-sovellusten kanssa ja salaisuuden luonnista seuraisi myöhemmin vain ongelmia. Clientin refresh tokenin vanhenemisaikaa voi halutessaan muuttaa, oletuksena se on 30 vuorokautta.

¹⁶ Create, Read, Update, Delete

Käyttäjään liittyvät tiedot, kuten sähköpostiosoite ja nimi, ovat tallessa User Poolissa. Koska on tarve tietää myös käyttäjään liittyvät roolit ja oikeudet, tulee luoda Identity Pool, joka linkitetään User Pooliin sekä App Clientiin. Kirjautunut käyttäjä saa automaattisesti ennalta määritellyt oikeudet.

5.2.3 GraphQL-rajapinta

Jos tietokantana olisi tarkoitus käyttää Amazonin DynamoDB:tä, AppSyncin GraphQL-skeema on mahdollista luoda joko tietokannan taulujen perusteella tai luontivelhon (wizard) avulla. Serverless-tietokannan tapauksessa työkaluja ei kuitenkaan voi käyttää, vaan kaikki on tehtävä itse.

Skeema on määritelmä siitä, missä muodossa data liikkuu palvelimen ja clientin, kuten mobiilisovelluksen, välillä. Skeema koostuu tyyppimäärittäyksistä (Type), kyselyistä (Query) ja mutaatioista (Mutation), joilla tehdään dataan muutoksia. Skeema, joka määrittää käyttäjätyyppin, kyselyn käyttäjän tiedoista ja uuden käyttäjän lisäämisen, voi näyttää esimerkiksi tällaiselta:

```
type User {  
  id: ID!  
  name: String!  
  email: AWSEmail!  
  createdAt: String!  
  updatedAt: String  
}  
type Query {  
  getUser: User  
}  
type Mutation {  
  createUser: User!  
}
```

Huutomerkki tyyppin perässä tarkoittaa pakollista kenttää, jonka arvo ei voi olla *null*. Skeema ei kuitenkaan yksistään riitä toimivan rajapinnan luomiseen, vaan lisäksi tarvitaan resolversiä, jotka määrittävät, kuinka erilaisiin kyselyihin vastataan. Mutaatioon createUser liittyvä uuden käyttäjän tietokantaan luova resolveri voi olla esimerkiksi muotoa

```
{
  "version": "2018-05-29",
  "statements": [
    "insert into databaseName.User (id,name,email) VALUES (:ID,
:NAME,:EMAIL)",
    "select * from databaseName.User WHERE id = :ID"
  ],
  "variableMap": {
    ":ID": "$ctx.identity.sub",
    ":NAME": "$ctx.identity.claims.name",
    ":EMAIL": "$ctx.identity.claims.email"
  }
}
```

Version päiväys riippuu tietokannan tyypistä ja on erilainen relaatiotietokannalle ja Lambda-funktioille, joita voi myöskin käyttää datalähteenä. Statements voi sisältää korkeintaan kaksi tietokantakyselyä ja rajoitus tulee AppSyncin puolelta. Tässä ensimmäinen kysely lisää käyttäjän tietokantaan ja toinen kysely hakee juuri lisätyn käyttäjän tiedot. createUser-mutaation ei tarvitse esitellyssä tapauksessa sisältää parametreja, koska \$context-muuttujan kautta pääsee käsiksi tiettyihin kirjautuneen käyttäjän tietoihin, jotka on tallennettu Cognito User Pooliin. Tämä tietenkin edellyttää, että rajapinnan autorisointimenetelmäksi on valittu AMAZON_COGNITO_USER_POOLS. \$context-muuttujan sisällöstä voi halutessaan lukea lisää viitteestä [97].

Helpointa olisi ollut, jos kaikki resolverit olisivat voineet olla suoraan yhteydessä tietokantaan. Muun muassa mutaatiot, jotka sisälsivät valinnaisena kenttänä viiteavaimen, eivät hyväksyneetkään kenttään *null*-arvoa, vaikka tietokanta olisi sen sallinut. Näissä tapauksissa resolverin olikin herätettävä Lambda-funktio, joka suoritti kyselyn tietokannalle. Nyt resolveri voikin olla muotoa

```
{
  "version": "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "arguments": $util.toJson($context.arguments.input),
    "userId": "$ctx.identity.sub"
  }
}
```

jossa kuormana voidaan välittää clientilta saatavien parametrien lisäksi myös muuta tarpeellista dataa, kuten kirjautuneen käyttäjän tunnistetieto.

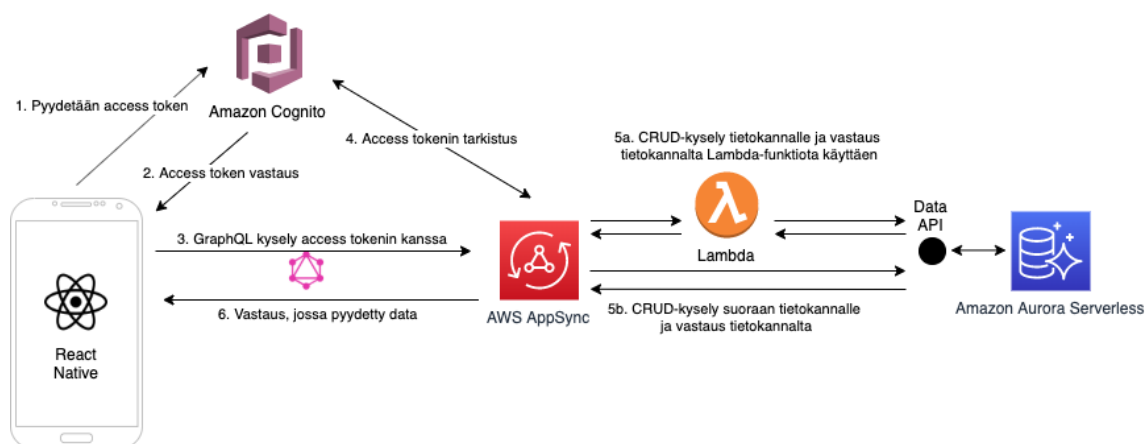
Lambda-funktiosta saadaan yhteys tietokannan Data API:n päätepisteeseen käyttämällä runtime-ympäristöstä riippuvaa kirjastoa. Node.js-runtime tarvitsee `aws-sdk`-kirjaston [98], kun taas Python-runtimeen on ladattava `boto3`-kirjasto [99]. Data API ei kuitenkaan ole Lambda-funktioiden kautta käytettynä täysin ongelmaton. Pientä keskeneräisyyttä selittänee sekin, että Data API julkaistiin vasta alkukesästä 2019 ja sitä kehitetään edelleen [95]. Data API:n Lambda-funktiolle palauttama vastaus voi olla esimerkiksi

```
{
  "numberOfRecordsUpdated": 0,
  "records": [
    [
      {
        "stringValue": "b477cd8a-c905-a17b-b0e2-ff5ed281525a",
      },
      {
        "stringValue": "John Doe",
      },
      {
        "stringValue": "john@example.com",
      },
      {
        "isNull": true
      }
    ]
  ]
}
```

Vastaus sisältää siis listan rivejä, jotka sisältävät listan sarakkeiden tyyppi-arvopareja. Hankaluus vastauksen tulkitsemisessa on se, että sarakkeiden nimien sijaan saadaankin niiden tyypit. Vastaus joudutaan joka kerta muuttamaan rajapinnan ymmärtämään muotoon ennen sen palauttamista rajapinnalle.

Kun rajapinta ja kaikki muut backendin osaset on saatu konfiguroitua ja koodattua, voidaan tarkastella, kuinka backend toimii yhteistyössä sovelluksen kanssa (kuva 11). Kuvan vaiheet 1. ja 2. suoritetaan vain sovellukseen kirjautuessa sekä access tokenia uusittaessa refresh tokenin avulla. Vaiheessa 3. sovellus tekee rajapinnalle pyynnön noutaa esimerkiksi käyttäjän tiedot tietokannasta. Pyyntöissä on määritelty, missä muodossa vastaus halutaan saada. Vaiheessa 4. AppSync tarkistaa pyynnön mukana saapuvan tokenin voimassaolon Cognitoilta. Jos token on oikeanlainen, AppSync suorittaa tietokantakyselyn joko suoraan

Data API:n kautta tai Lambda-funktiota apuna käyttäen (vaihe 5a tai 5b). Vaiheessa 6. AppSync palauttaa sovelluksen pyytämän datan halutussa muodossa.



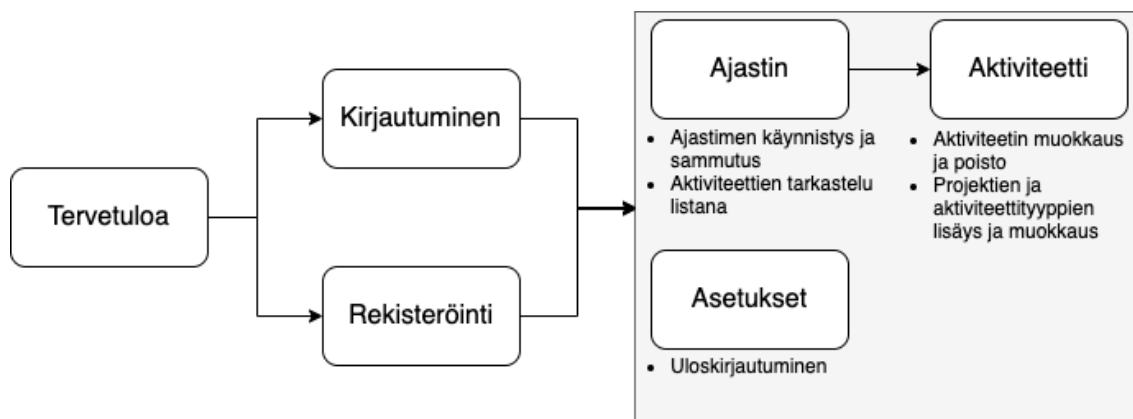
Kuva 11. Kaavio datan liikkumisesta mobiilisovelluksen ja backendin välillä.

5.3 Sovelluksen frontend

Kuten jo aikaisemmin on tullut esille, sovelluksen frontend on koodattu käyttäen React Native -kirjastoa. Koska kirjastoja kehitetään jatkuvasti, mainitaan käytetyn kirjaston kohdalla myös versio, jota opinnäytetyössä on käytetty. Kirjoitushetkellä käytössä oli `react@16.8.6` ja `react-native@0.60.5`. JavaScriptin sijaan käytettiin TypeScriptiä, joka sisältää muuttujien tyyppimäärittelyt. Luvussa kuvaillaan sovelluksen toiminnallisuuksia keskittyen kuitenkin siihen, kuinka frontend kommunikoi backendin kanssa.

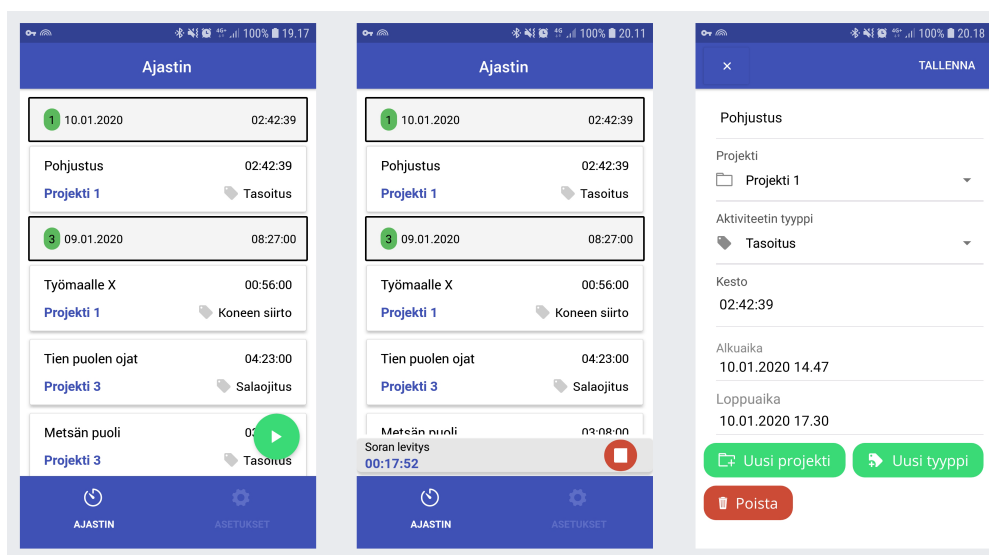
5.3.1 Rakenne

Sovelluksen rakennetta selventää kuva 12. Kun käyttäjä avaa sovelluksen siten, ettei ole jo valmiiksi kirjautuneena, hän näkee ensimmäisenä Tervetuloa-ruudun. Ruudulta voi valita joko kirjautumisen tai uutena käyttäjänä rekisteröitymisen. Rekisteröityvää käyttäjää pyydetään antamaan sähköpostiosoite, johon järjestelmä lähettää vahvistuskoodin. Koodi on voimassa 24 tuntia. Sähköpostiosoitteiden tulee olla uniikkeja, eli samalla osoitteella on mahdollista rekisteröityä vain yhden kerran, koska sähköpostiosoite toimii sovelluksen käyttäjätunnuksena.



Kuva 12. Työajanseurantasovelluksen rakenne.

Käyttäjä pääsee sisälle sovellukseen, kun on suorittanut onnistuneesti kirjautumisen tai rekisteröinnin. Selattavia välilehtiä on kaksi: Ajastin ja Asetukset. Kuvassa 13 on havainnollistava kollaasi Ajastin-välilehden toiminnoista. Välilehdellä voi tarkastella listana aikaisemmin tallennettuja aktiviteetteja ja aloittaa uuden tallennuksen vihreästä play-painikkeesta (näkymä vasemmalla). Käynnissä olevan aktiviteetin tiedot näkyvät alhaalla ja kellon voi pysäyttää stop-painikkeella (keskimmäinen näkymä). Mitä hyvänsä listan aktiviteettia tai käynnissä olevaa aktiviteettia painamalla avautuu muokkausmodaali (näkymä oikealla), josta voi asettaa aktiviteetille kuvauksen, projektin sekä tyypin. Modaalin kautta on mahdollista lisätä ja muokata projekteja sekä aktiviteetin tyyppejä. Valitun aktiviteetin voi myös poistaa. Asetukset-välilehdeltä käyttäjä voi kirjautua ulos sovelluksesta.



Kuva 13. Ajastin-välilehden näkymät (vasemmalla ja keskellä) sekä aktiviteetin muokkausmodaali (oikealla).

5.3.2 Yhteys backendiin

Jotta frontendistä saadaan yhteys backendiin, on ladattava oikeat kirjastot ja tallennettava asetukset. Yksi tärkeimmistä kirjastoista on `aws-amplify`, jonka voi ladata kokonaisena, jolloin paketti on suurempi, tai yksittäin pienempinä moduuleina. Tarpeellisia moduuleita ovat muun muassa `@aws-amplify/core@1.1.2` ja `@aws-amplify/auth@1.3.3`. Myös kirjasto `amazon-cognito-identity-js@3.0.15` tarvitaan.

`Aws-amplify`ta varten sovelluksesta on löydyttävä tiedosto, joka sisältää täydennettynä seuraavat tiedot

```
const awsconfig = {
  Auth: {
    // Amazon Cognito Identity Pool ID
    identityPoolId: "XX-XXXX-X:XXXXXXXX-XXXX-1234-abcd-1234567890ab",
    // Amazon Cognito alue
    region: "XX-XXXX-X"
    // Amazon Cognito User Pool ID
    userPoolId: "XX-XXXX-X_abcd1234 ",
    // Amazon Cognito Web Client ID, 26-merkkiä
    userPoolWebClientId: "a1b2c3d4e5f6g7h8i9j0k1l2m3",
  }
}
export default awsconfig;
```

jotta sovelluksen juuritiedostossa, kuten `App.tsx`, voidaan kutsua

```
import Amplify from "@aws-amplify/core";
import awsConfig from "../awsConfig"
Amplify.configure(awsconfig);
```

jolloin saadaan autentikointiin tarvittavat tiedot käyttöön koko sovelluksen sisällä. *Tunnisteet sisältävää tiedostoa ei tule tallentaa versionhallintaan turvallisuussyistä.*

Autentikointiin liittyviä tarpeellisia funktioita ovat:

`signUp()` – rekisteröi käyttäjän

`confirmSignUp()` – vahvistaa rekisteröinnin vahvistuskoodilla

`resendSignUp()` – yrittää uudelleen rekisteröintiä, lähettää uuden koodin
`signIn()` – sisäänkirjaa käyttäjän
`currentSession()` – uusii autentikointi-tokenin
`currentAuthenticatedUser()` – palauttaa kirjautuneena olevan käyttäjän tiedot
`signOut()` – uloskirjaa käyttäjän

Yllä olevat funktiot saa käyttöön sen jälkeen, kun tiedostossa on otettu `import-` komennolla käyttöön moduuli `@aws-amplify/auth`.

AppSyncin GraphQL-rajapintaa varten on taas tallennettava rajapintaan liittyvät tiedot. Tiedoston sisältö voi olla vaikka

```
export default {
  ApiUrl: "https:// a1b2c3d4e5f6g7h8i9j0k1l2m3.rajapinnan-nimi.XX-XXXX-X.amazonaws.com/graphql",
  Region: "XX-XXXX-X",
  AuthMode: "AMAZON_COGNITO_USER_POOLS",
};
```

Rajapinnalle tarpeellisista kirjastoista on asennettava `aws-appsync@2.0.2`, `aws-appsync-react@2.0.2`, `react-apollo@2.5.8`, sekä `graphql-tag@2.10.1`. Jotta heti sovelluksen käynnistyessä luotaisiin AppSync-client, jonka kautta rajapintayhteys muodostetaan, on juuritiedostoon kirjoitettava seuraavat rivit:

```
import AWSAppSyncClient from "aws-appsync";
import { Rehydrated } from "aws-appsync-react";
import { ApolloProvider } from "react-apollo";
import { MyRootComponent } from "./myRootComponent";
import AppSync from "./appsync-conf";

const client = new AWSAppSyncClient({
  url: AppSync.ApiUrl,
  region: AppSync.Region,
  disableOffline: true,
  auth: {
    type: AppSync.AuthMode,
    jwtToken: async () => (await Auth.currentSession()).getIdToken().getJwtToken(),
  },
});
```



```

class App extends React.Component {
  public render() {
    return (
      <ApolloProvider client={client}>
        <Rehydrated>
          <MyRootComponent />
        </Rehydrated>
      </ApolloProvider>
    );
  }
}

export default App;

```

Client saa asetuksikseen aikaisemmin määritellyn urlin, alueen, autentikointitavan sekä JWT-tokenin. Token ei ole välttämättä tiedossa heti sovelluksen avautuessa, kuten tilanteessa, jossa käyttäjä ei ole kirjautuneena sisälle. Tämän vuoksi token haetaan asynkronisesti ja tallennetaan sitten, kun se on saatavilla.

Luvusta 5.2.3 muistetaan, että rajapinnalle menevät pyynnöt ovat joko muotoa Query tai Mutation. Sovelluksen kannalta tarpeelliset pyynnöt kannattaa tallentaa omiksi tiedostoikseen, jotta muu koodi pysyy siistimpänä ja pyynnöt ovat uudelleenkäytettävissä ilman turhaa uudelleenkirjoitusta. Tiedoston sisältö voi olla esimerkiksi

```

import gql from "graphql-tag";

export default gql`
  mutation createActivity($startDateTime: String!) {
    createActivity(input: { startDateTime: $startDateTime }) {
      id
      description
      startDateTime
      endDateTime
      projectId
      activityTypeId
    }
  }
`;

```

Tiedostossa määritellyn mutaation tai kyselyn tulee noudattaa samaa skeemaa kuin mitä käytetään rajapinnassa. Mutaatio- ja kyselypyynnöt toteutetaan hieman eri tavoilla

```

import moment from "moment";
import createActivity from "../mutations/CreateActivity";
import getUser from "../queries/GetUser";
import { graphql, compose, withApollo } from "react-apollo";

export class ExampleComponent extends React.Component{

  private async queryUser () {
    await this.props.client
      .query({
        query: getUser,
      })
      .then(async ({ data }: { data: any }) => {
        /** Do something with data */
      })
      .catch(async (err: any) => {
        /** Oops, error */
      });
  }

  private async newActivity() {
    const now = moment.utc().format("YYYY-MM-DD HH:mm:ss");
    await this.props
      .createActivity({ variables: { startDateTime: now } })
      .then(({ data }: { data: any }) => {
        /** Do something with data */
      })
      .catch((err: any) => {
        /** Oops, error */
      });
  }
}

export default compose(
  withApollo,
  graphql(createActivity, { name: "createActivity" }),
)(ExampleComponent);

```

Esimerkistä on hyvä huomata export-komentoon liittyvät rivit. `withApollo` tarjoaa propsien kautta suoran pääsyn AppSync clientin query-funktioon ja `graphql`-funktion sisään paketoitu `createActivity`-mutaatio löytyy myös propseista. Näin funktiot `queryUser()` ja `newActivity()` voidaan suorittaa silloin, kun on sopiva hetki eivätkä pyynnöt lähde heti, kun komponentti luodaan, toisin kuin `react-apollo`-kirjaston omaa `Mutation`-komponenttia käyttämällä [102].

Aina, kun rajapintaan on tarve saada yhteys, tehdään se edellä kuvaillulla tavalla. Kun käyttäjä on kirjautunut sisään sovellukseen, noudetaan kaikki tallennetut aktiviteetit. Kun kellotus alkaa, luodaan uusi aktiviteetti. Kun kello pysäytetään, päivitetään aktiviteetin loppuaika. Kun aktiviteetin nimeä muutetaan, liitetään aktiviteetti johonkin projektiin tai annetaan sille luokka, laitetaan päivityspyyntö. Kaikki tietokannan kanssa tapahtuvat toimet tehdään rajapinnan kautta.

Tässä vaiheessa on tärkeää, että verkkoyhteys toimii tai kaikista pyynnöistä tulee virheviesti. AppSync sisältää sisäänrakennettuna offline-tuen, mutta toistaiseksi se on sovelluksessa pois käytöstä. On tunnettava ominaisuuden toiminta, jotta sitä voi käyttää tehokkaasti ja välttää yllättäviä tilanteita, joissa asiat eivät toimiakaan niin kuin pitäisi.

6 Pohdinta

Opinnäytetyön viimeisessä luvussa käydään läpi, miten projektin toteutus sujui, mitä olisi voitu tehdä toisin ja mitä jäi vielä tekemättä.

6.1 Kehitysprosessi, työkalut ja kustannukset

Opinnäytetyön parasta antia oli uusiin teknologioihin tutustuminen ja uuden oppiminen. Oli ennakkoon tiedossa, että backend tulee vaatimaan työhön käytettävästä ajasta leijonanosan, koska opittavaa oli niin paljon. En ollut aikaisemmin käyttänyt mitään Amazon Web Servicen palveluita ja backendiä olin pyörittänyt vain paikallisesti omalla koneella.

Alun perin suunnittelin käyttäväni REST-rajapintaa, eli Amazon API Gatewayta, koska se oli minulle GraphQL-rajapintaa tutumpi vaihtoehto. Opinnäytetyön teoriaosia kirjoittaessani muutin kuitenkin mieltäni, koska AppSync tukee offline-tilaa, joka on tarpeellinen silloin, kun sovellusta käytetään paikoissa, joissa ei ole verkkoyhteyttä.

Kaiken kaikkiaan olin tyytyväinen valitsemiini AWS:n palveluihin ja lopputulokseen. Tiesin, että haasteita tulee erityisesti tietokannan kylmäkäynnistyksestä, joka vie aikaa noin 45–60 sekuntia. Tietokanta kannattaa siis laittaa käynnistymään niin aikaisessa vaiheessa kuin mahdollista. Frontendin kehityksen aikana huomasin, että ainoastaan kirjautunut käyttäjä voi tehdä AppSyncille menevän tietokannan herätyskutsun, koska käyttämilläni asetuksilla AppSync vaatii kutsujen mukana voimassa olevan access tokenin. Tietokantaa ei siis saa hereille vielä silloin, kun uusi käyttäjä täyttää rekisteröintilomaketta. Tämä tulisi korjata myöhemmin joko niin, että sovellus kutsuu suoraan tietokannan herättävää Lambda-funktiota tai AppSyncissä otetaan Cognito-tunnistuksen lisäksi käyttöön myös API key -autorisointi herätyskyselyä varten.

Kehittäjä ja toimeksiantaja tietävät, että tietokanta sammutetaan kustannussyistä ja sen vuoksi käynnistymistä joutuu odottamaan kohtalaisen pitkältä tuntuvan ajan. Sovelluksen käyttöä voisi sujuvoittaa sillä, että tietokanta ajastetaan pysymään päällä silloin, kun sillä on ennakoitavaa käyttöä. Ajastus voisi olla arkisin kello 6 ja 17 välillä tai vielä kustannustehokkaammin aamulla, päivän tauon aikoihin ja illalla, kun työt normaalisti lopetetaan.

Jos sovellus olisi julkisesti käytössä, käyttäjäkokemus kärsisi pitkistä odotusajoista. Sovelluksesta maksava käyttäjä haluaa käytön olevan sujuvaa. Tässä tapauksessa tuotantoympäristössä pitäisi olla tietokanta, joka olisi jatkuvasti saatavutettavissa ja valinta olisi joku muu kuin serverless.

Sovelluksen kehitysprosessin ajan Amazon Aurora Serverless MySQL -tietokannasta tuli kuluja noin 5–10 euroa kuukaudessa. Tässä vaiheessa voidaan siis arvioida, että säännöllinen käyttö tulisi maksamaan noin 10 euroa per kuukausi. Myöhemmässä vaiheessa, kun tietokannan koko kasvaa, myös kulut kasvavat.

Sovelluksesta saatiin opinnäytetyön aikana valmiiksi prototyyppi, jota pystyy käyttämään, mutta joka ei sellaisenaan vielä vastaa aloituspalaverissa asetettuja valmiin sovelluksen kriteerejä. On siis pohdittava, saako toimeksiantaja sovelluksen viimeistelyyn vaadittaville rahoille vastinetta valmiin sovelluksen tuomina

hyötyinä vai onko järkevämpää valita jo markkinoilla olevista vaihtoehtoista sopivin ja maksaa siitä käytön mukaan.

6.2 Jatkokehitys

Kuten luvussa 5.1 jo mainittiin, kaksi selkeää jatkokehityskohdetta ovat opinnäytetyön ulkopuolelle jääneet raportointi- ja ajopäiväkirjaominaisuus. Näiden lisäksi sähköisessä ajopäiväkirjassa tarvittavaa paikannusta ja karttaa voi hyödyntää myös työajanseurannassa.

Sovellukseen olisi mahdollista lisätä automaattinen ominaisuus luoda viikko- ja kuukausiraportit pdf-muodossa kaikista tallennetuista tunneista ja lähettää ne sähköpostiin. Käyttäjän on myös itse tarve luoda haluamanlaisiaan raportteja valitsemallaan aikavälillä tietyistä projekteista tai aktiviteettityypeistä. Raportit voisi joko lähettää sähköpostiin tai ladata laitteelle. Näitä raportteja on mahdollista liittää lähetettäviin laskuihin todentamaan asiakkaalle tehdyt tunnit.

Kun yrittäjä käyttää ajoneuvoaan sekä työajoihin että omiin ajoihin, verottaja vaatii merkitsemään ajankohdan, ajetut kilometrit sekä matkan tarkoituksen ajopäiväkirjaan. Sovellus voisi paikanninta apuna käyttäen tallentaa käyttäjän sijainnin ja laskea kuljetun matkan. Sovelluksen voisi lisätä kartan, josta näkee kuljetun reitin sekä reitin tiedot.

Kun sovellus sisältäisi kartan, myös projektin sijainnit voisi merkitä kartalle. Projekteille voisi määrittää alueen, jonka sisällä kyseiseen projektiin liittyvät työt suoritetaan. Kun sovellus havaitsee käyttäjän olevan merkityn alueen sisällä, projekti valittaisiin automaattisesti, jolloin käyttäjän ei tarvitsisi työtä aloittaessaan itse vaihtaa projektia.

Nykyisellään sovellus vaatii uutta käyttäjää rekisteröitymään ja luomaan uudet tunnukset. Sovellukseen voisi toteuttaa kirjautumisen federoidun julkisen palveluntarjoajan, kuten Googlen tai Facebookin, tunnusten avulla. Käyttäjä hyötyy, kun voi käyttää olemassa olevia tunnuksiaan.

Tällä hetkellä sovellus ei näytä käyttäjälle ilmoituksia. Push notifikaatioita voisi hyödyntää silloin, kun sovellus havaitsee työajan kellotuksen jääneen päälle pitkäksi aikaa. Sovellus voisi kysyä, onko kello jäänyt vahingossa sammuttamatta.

Ylipäättään sovellusta tulisi kehittää yhteistyössä käyttäjien kanssa, jolloin saadaan arvokasta palautetta sovelluksen toimivuudesta ja käytettävyydestä sekä vinkkejä siitä, mitä voisi olla lisää tai tarve tehdä paremmin. Sovellus on kuitenkin olemassa käyttäjiään varten.

Lähteet

1. Clement, J. 2019. Number of apps available in leading app stores as of 3rd quarter 2019. Statista. <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>. 23.11.2019.
2. Clement, J. 2019. Combined global Apple App Store and Google Play app downloads from 1st quarter 2015 to 2nd quarter 2019 (in billions). Statista. <https://www.statista.com/statistics/604343/number-of-apple-app-store-and-google-play-app-downloads-worldwide/>. 23.11.2019.
3. Moogk, D. R. 2012. Minimum viable product and the importance of experimentation in technology startups. *Technology Innovation Management Review* vol. 2, no. 3, 23-26.
4. Chen, P. M. & Noble, B. D. 2001. When virtual is better than real. *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, 133-138.
5. Merkel, D. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* vol. 239 2.
6. Roberts, M. 2018. Serverless architectures: what is serverless?. <https://martinfowler.com/articles/serverless.html>. 23.11.2019.
7. Hendrickson, S. ym. 2016. Serverless computation with openlambda. 8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16).
8. Pettey, C. 2019. 6 Best Practices for Creating a Container Platform Strategy. Gartner. <https://www.gartner.com/smarterwithgartner/6-best-practices-for-creating-a-container-platform-strategy/>. 23.11.2019.
9. Raj, P., Chelladurai, J. S. & Singh, V. 2015. *Learning Docker*. Birmingham – Mumbai: Packt Publishing Ltd.
10. IBM Cloud Education. 2019. Containerization. <https://www.ibm.com/cloud/learn/containerization>. 23.11.2019.
11. Google Trends. 2019. Docker. <https://trends.google.com/trends/explore?date=2013-01-01%202019-11-25&q=docker>. 25.11.2019.
12. Bhardwaj, S., Jain, L. & Sandeep, J. 2010. Cloud computing: A study of infrastructure as a service (IAAS). *International Journal of engineering and information Technology*, vol. 2 (1), 60-63.
13. Microsoft Azure. 2019. What is PaaS? <https://azure.microsoft.com/en-us/overview/what-is-paas/>. 23.11.2019.
14. Fromm, K. 2012. Why The Future Of Software And Apps Is Serverless. <https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/>. 23.11.2019.
15. AWS. 2019. AWS Lambda Pricing. <https://aws.amazon.com/lambda/pricing/>. 23.11.2019.
16. Brown, R. S. 2015. JAWS Is Now Serverless. *Serverless Code*. <https://serverlesscode.com/post/serverless-formerly-jaws/>. 23.11.2019.
17. Serverless. 2019. Github – Serverless. <https://github.com/serverless/serverless>. 23.11.2019.
18. Google Trends. 2019. Serverless. <https://trends.google.fi/trends/explore?date=2015-01-01%202019-08-31&q=Serverless>. 23.11.2019.
19. AWS. 2019. AWS Lambda – Serverless Compute. <https://aws.amazon.com/lambda/>. 23.11.2019.
20. Google Cloud. 2019. Cloud Functions – Event-driven Serverless Computing. <https://cloud.google.com/functions/>. 23.11.2019.

21. Microsoft Azure. 2019. Azure Functions – Develop Faster With Serverless Compute. <https://azure.microsoft.com/en-us/services/functions/>. 23.11.2019
22. IBM Cloud. 2019. IBM Cloud Functions. <https://cloud.ibm.com/functions/>. 23.11.2019.
23. Ellis, A. 2018. Introducing stateless microservices for OpenFaaS. OpenFaaS. <https://www.openfaas.com/blog/stateless-microservices/>. 23.11.2019.
24. Google Cloud. 2019. Cloud Functions Execution Environment. <https://cloud.google.com/functions/docs/concepts/exec>. 23.11.2019.
25. AWS. 2019. AWS Lambda Limits. <https://docs.aws.amazon.com/lambda/latest/dg/limits.html>. 23.11.2019.
26. AWS. 2019. Amazon ESC – Run containerized applications in production. <https://aws.amazon.com/ecs/>. 23.11.2019.
27. AWS. 2019. AWS Fargate – Run containers without having to manage servers or clusters. <https://aws.amazon.com/fargate/>. 23.11.2019.
28. AWS. 2019. Run a Serverless "Hello, World!" with AWS Lambda. <https://aws.amazon.com/getting-started/tutorials/run-serverless-code/>. 23.11.2019.
29. AWS. 2019. Amazon EC2 Reserved Instances. <https://aws.amazon.com/ec2/pricing/reserved-instances/>. 23.11.2019.
30. Google Cloud. 2019. Pricing. <https://cloud.google.com/functions/pricing>. 23.11.2019.
31. AWS. 2019. Amazon Cognito – Simple and Secure User Sign-Up, Sign-In, and Access Control. <https://aws.amazon.com/cognito/>. 23.11.2019.
32. AWS. 2019. Amazon Aurora Pricing. <https://aws.amazon.com/rds/aurora/pricing/>. 23.11.2019.
33. IBM Cloud. 2019. IBM Cloud Functions – Pricing. <https://cloud.ibm.com/functions/learn/pricing>. 23.11.2019.
34. Docker docs. 2019. Scale the service in the swarm. <https://docs.docker.com/engine/swarm/swarm-tutorial/scale-service/>. 23.11.2019.
35. Kubernetes. 2019. Horizontal Pod Autoscaler. <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. 23.11.2019.
36. AWS. 2019. Serverless Computing. <https://aws.amazon.com/serverless/>. 23.11.2019.
37. Kane, S. P. & Matthias, K. 2018. Docker: Up & Running: Shipping Reliable Containers in Production. Sebastopol: O'Reilly Media.
38. Benítez, L. H. Life and death of a container. Medium. <https://medium.com/devopsion/life-and-death-of-a-container-146dfc62f808>. 23.11.2019.
39. Jackson, D. & Clynch, G. 2018. An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions. IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 154-160.
40. Garfield, D. 2017. Testing Strategies for Docker Driven Development. Codefresh. <https://codefresh.io/docker-tutorial/testing-strategies-for-docker/>. 23.11.2019.
41. Firebase. 2019. Run functions locally. <https://firebase.google.com/docs/functions/local-emulator>. 23.11.2019.

42. Microsoft Azure. 2019. Work with Azure Functions Core Tools.
<https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local>. 23.11.2019.
43. AWS. 2019. Testing and Debugging Serverless Applications.
<https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-test-and-debug.html>. 23.11.2019.
44. Buckholz, G. 2019. How to solve the testing challenges that come with serverless apps. TechBeacon. <https://techbeacon.com/app-dev-testing/how-solve-testing-challenges-come-serverless-apps>. 23.11.2019.
45. Cui, Y. 2019. How to Test Serverless Apps. Epsagon.
<https://epsagon.com/blog/how-to-test-serverless-apps/>. 23.11.2019.
46. Majors, C. 2016. Operational best practices #serverless.
<https://charity.wtf/2016/05/31/operational-best-practices-serverless/>. 23.11.2019.
47. Shu, R., Gu, X. & Enck, W. 2017. A study of security vulnerabilities on docker hub. Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, 269-280.
48. Rahic, A. 2019. Fantastic Serverless security risks, and where to find them. Serverless Blog. <https://serverless.com/blog/fantastic-serverless-security-risks-and-where-to-find-them/>. 23.11.2019.
49. Richter, F. 2019. Amazon Leads the Race to the Cloud. Statista.
<https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/>. 23.11.2019.
50. IronFunctions. 2019. Open Source Serverless Computing.
<https://open.iron.io/>. 23.11.2019.
51. OpenLambda. 2019. Github – OpenLambda. <https://github.com/open-lambda/open-lambda>. 23.11.2019.
52. OpenFaaS. 2019. Serverless Functions, Made Simple.
<https://www.openfaas.com/>. 23.11.2019.
53. Webtask. 2019. All you need is code. <https://webtask.io/>. 23.11.2019.
54. Block, G. 2018. We Are Sunsetting Extend. Auth0.
<https://auth0.com/blog/we-are-sunsetting-extend/>. 23.11.2019.
55. AWS. 2019. Cloud Products. <https://aws.amazon.com/products/>. 23.11.2019.
56. Hecht, L. E. 2017. AWS Lambda Still Towers Over the Competition, but for How Much Longer? The New Stack. <https://thenewstack.io/aws-lambda-still-towers-competition-much-longer/>. 23.11.2019.
57. AWS. 2019. AWS Identity and Access Management (IAM).
<https://aws.amazon.com/iam/>. 23.11.2019.
58. AWS. 2019. AWS Lambda Features.
<https://aws.amazon.com/lambda/features/>. 23.11.2019.
59. AWS. 2015. Introducing Amazon API Gateway.
<https://aws.amazon.com/about-aws/whats-new/2015/07/introducing-amazon-api-gateway/>. 23.11.2019.
60. AWS. 2019. Amazon API Gateway Features. <https://aws.amazon.com/api-gateway/features/>. 23.11.2019.
61. Walker, T. 2017. Introducing AWS AppSync – Build data-driven apps with real-time and off-line capabilities. AWS.
<https://aws.amazon.com/blogs/aws/introducing-amazon-appsync/>. 23.11.2019.

62. How to GraphQL. 2019. GraphQL is the better REST.
<https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>. 23.11.2019.
63. AWS. 2019. AWS AppSync Features.
<https://aws.amazon.com/appsync/product-details/>. 23.11.2019.
64. AWS. 2019. AWS AppSync Overview. <https://aws.amazon.com/appsync/>. 23.11.2019.
65. AWS. 2019. Amazon Aurora Serverless.
<https://aws.amazon.com/rds/aurora/serverless/>. 23.11.2019.
66. AWS. 2019. Using Amazon Aurora Serverless.
<https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless.html>. 23.11.2019.
67. McDonald, P. 2008. Introducing Google App Engine + our new blog. Google App Engine Blog.
<http://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html>. 23.11.2019.
68. Google Cloud. 2019. Products and services.
<https://cloud.google.com/products/>. 23.11.2019.
69. Polites, J. 2017. Google Cloud Functions: a serverless environment to build and connect cloud services. Google Cloud.
https://cloud.google.com/blog/products/gcp/google-cloud-functions-a-serverless-environment-to-build-and-connect-cloud-services_13. 23.11.2019.
70. Google Cloud. 2019. Cloud Functions Overview.
<https://cloud.google.com/functions/docs/concepts/overview>. 23.11.2019.
71. Google Cloud. 2019. HTTP Triggers.
<https://cloud.google.com/functions/docs/calling/http>. 23.11.2019.
72. Google Cloud. 2019. Google Cloud Functions Pricing.
<https://cloud.google.com/functions/pricing>. 23.11.2019.
73. DeMichillie, G. 2014. Welcome Firebase to the Google Cloud Platform Team. Google Cloud Platform Blog.
<https://cloudplatform.googleblog.com/2014/10/welcome-firebase-to-google-cloud-platform.html>. 23.11.2019.
74. Firebase. 2019. Firebase Authentication.
<https://firebase.google.com/docs/auth/>. 23.11.2019.
75. Firebase. 2019. Firebase Products. <https://firebase.google.com/products>. 23.11.2019.
76. Firebase. 2019. Access data offline.
<https://firebase.google.com/docs/firestore/manage-data/enable-offline>. 23.11.2019.
77. Kotwani, S. 2018. The Firebase Blog - Introducing ML Kit for Firebase. Firebase. <https://firebase.googleblog.com/2018/05/introducing-ml-kit-for-firebase.html>. 23.11.2019.
78. Kiriathy, Y. 2016. Announcing general availability of Azure Functions. Microsoft Azure. <https://azure.microsoft.com/en-us/blog/announcing-general-availability-of-azure-functions/>. 23.11.2019.
79. Azure. 2019. Github – Azure-Functions. <https://github.com/Azure/azure-functions>. 2019.
80. Microsoft Azure. 2019. Azure Functions runtime versions overview.
<https://docs.microsoft.com/en-us/azure/azure-functions/functions-versions>. 22.11.2019.

81. IBM. 2019. Serverless Computing.
https://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=9368. 23.11.2019.
82. IBM Cloud. 2019. IBM Cloud Functions / FAQ.
<https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-faq>. 23.11.2019.
83. IBM Cloud. 2019. IBM Cloud Functions / Monitoring Activity.
<https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-monitor>. 23.11.2019.
84. IBM Cloud. 2019. IBM Cloud Functions / Viewing logs.
https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-logs#logs_logdna. 23.11.2019.
85. IBM Cloud. 2019. Cloudant.
<https://cloud.ibm.com/catalog/services/cloudantNoSQLDB#about>. 23.11.2019.
86. IBM Cloud. 2019. App ID.
<https://cloud.ibm.com/catalog/services/AppID#about>. 23.11.2019.
87. IBM Cloud. 2019. Build with Watson - The AI platform for business.
<https://cloud.ibm.com/developer/watson/dashboard>. 23.11.2019.
88. AWS. 2019. Limits in Amazon Cognito.
<https://docs.aws.amazon.com/cognito/latest/developerguide/limits.html>. 28.12.2019.
89. AWS. 2019. AWS Secrets Manager.
<https://aws.amazon.com/secrets-manager/>. 28.12.2019.
90. React Native. 2019. <https://facebook.github.io/react-native/>. 31.12.2019.
91. Octoverse. 2018. Repositories. <https://octoverse.github.com/2018/projects.html#repositories>. 31.12.2019.
92. React. 2019. Introducing JSX. <https://reactjs.org/docs/introducing-jsx.html>. 31.12.2019.
93. React Native. 2019. State. <https://facebook.github.io/react-native/docs/state>. 31.12.2019.
94. Redux. 2019. <https://redux.js.org>. 31.12.2019.
95. MobX. 2019. <https://mobx.js.org>. 31.12.2019.
96. AWS. 2019. Amazon Simple Email Service. <https://aws.amazon.com/ses/>. 28.12.2019.
97. AWS. 2019. Resolver Mapping Template Context. <https://docs.aws.amazon.com/appsync/latest/devguide/resolver-context-reference.html>. 30.12.2019.
98. AWS. 2019. Github – aws-sdk-js. <https://github.com/aws/aws-sdk-js>. 30.12.2019.
99. Boto. 2019. Github – boto3. <https://github.com/boto/boto3/>. 30.12.2019.
100. Barr, J. 2019. New – Data API for Amazon Aurora Serverless. AWS.
<https://aws.amazon.com/blogs/aws/new-data-api-for-amazon-aurora-serverless/>. 30.12.2019.
101. ERDPlus. 2020. <https://erdplus.com/>. 1.1.2020.
102. Apollo GraphQL. 2020. Mutations. <https://www.apollographql.com/docs/react/v2.5/essentials/mutations/>. 7.1.2020.