LAMK
Lahden ammattikorkeakoulu
Lahti University of Applied Sciences

FAAS-PALVELUUN POHJAUTUVAN TAUSTAJÄRJESTELMÄN KEHITYS

Case: HiQ Finland Oy

LAHDEN AMMATTIKORKEAKOULU
Insinööri (AMK)
Tieto- ja viestintätekniikka
Syksy 2018
Andreas Siivola

Tiivistelmä

Tekijä(t)	Julkaisun laji	Valmistumisaika
Siivola, Andreas	Opinnäytetyö, AMK	Syksy 2018
	Sivumäärä	
	41	

Työn nimi

FaaS-palveluun pohjautuvan taustajärjestelmän kehitys
Case: HiQ Finland Oy

Tutkinto

Insinööri (AMK)

Tiivistelmä

Opinnäytetyön tavoitteena oli toteuttaa ja testata FaaS-palveluun pohjautuva taustajärjestelmä osaksi aktiivisessa kehityksessä olevan valvontapalvelun arkkitehtuuria. Opinnäytetyön toimeksiantaja oli HiQ Finland Oy.

Työssä käytettiin pilvilaskenta-alustana Amazon Web Servicesiä. Alustan tarjoamista palveluista keskityttiin erityisesti alustan FaaS-palveluun Lambdaan ja sitä tukeviin API Gatewayn ja CloudWatchin ominaisuuksiin.

Taustajärjestelmän kehitykseen käytettiin apuna Serverless Framework -työkaluohjelmistoa. Työssä käydään läpi työkaluohjelmiston toimintaperiaatteita ja funktioon valitun arkkitehtuurimallin vaikutuksia tarvittavien funktioiden lukumäärään ja käsittelijäfunktioiden kompleksisuuteen.

Työn tavoitteeseen päästiin ja tuloksena oli toimiva taustajärjestelmä. Taustajärjestelmää ei kuitenkaan otettu käyttöön osana valvontapalvelun arkkitehtuuria, sillä FaaS-palvelun funktioiden suoritusajan optimointia rajoittavana tekijänä on ulkopuolinen rajapinta, josta suuri osa taustajärjestelmän rajapinnan päätepisteistä noutaa tietojaan. Kehitystyön aikana ei tullut vastaan ylitsepääsemättömiä ongelmia. Sen aikana huomattiin, kuinka vaivatonta Serverless Framework tekee serverless-sovellusten kehityksestä.

Asiasanat

serverless, Amazon Web Services, Node.js, FaaS, BaaS, Serverless Framework, Lambda

Abstract

Author(s)	Type of publication	Published
Siivola, Andreas	Bachelor's thesis	Autumn 2018
	Number of pages	
	41	

Title of publication

FaaS-based back-end development
Case: HiQ Finland Oy

Name of Degree

Bachelor of Engineering

Abstract

The objective of the thesis was to implement and test a serverless compute-based back end as a part of the architecture of a monitoring service currently in active development. The Bachelor's thesis was commissioned by HiQ Finland Oy.

The Amazon Web Services cloud computing platform was used in the work. Special emphasis was put on Lambda, the platforms FaaS service, and its supporting features from API Gateway and CloudWatch.

The Serverless Framework toolkit was used to speed up the development process significantly. The thesis examines the operating principles of the toolkit as well as the effects that the chosen architectural pattern has on the number of functions needed for the project and the complexity of the handler functions.

The objective of the thesis was achieved, and the result was a working back end. The back end was not implemented as a part of the monitoring service architecture because of the limited ability to optimize the functions. The limitation is caused by an external API that is called from the back end. During the development process it was noted how effortless it is to create serverless applications using the Serverless Framework.

Keywords

serverless, Amazon Web Services, Node.js, FaaS, BaaS, Serverless Framework, Lambda

SISÄLLYS

1 JOHDANTO

Serverless-teknologia on muodostunut viime aikoina eräänlaiseksi muotisanaksi web-ohjelmistokehityksessä ja verkkopalveluissa. Serverless-palveluita on käytettävissä moniin erilaisiin käyttötarkoituksiin. Käyttämällä kyseisiä palveluita voidaan vähentää kehitystyöhön kulutettua aikaa ja operaationaalista vastuuta sovelluksen skaalauksesta sen saatavuuteen.

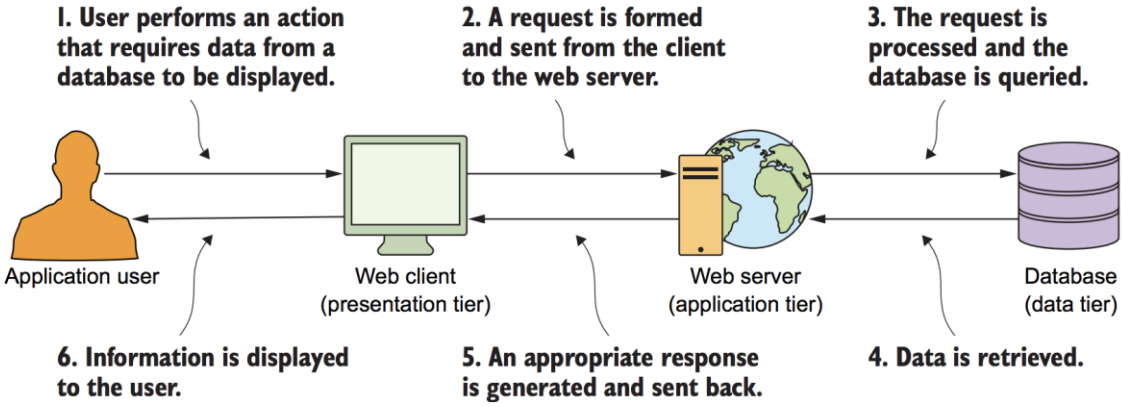
Opinnäytetyön toimeksiantajana oli HiQ Finland Oy. HiQ Finland on osa HiQ-konsernia. Konserni työllistää yhteensä 1600 osaajaa, joista lähes 300 työskentelee Suomessa. HiQ Finland on erikoistunut vaativiin mobiili- ja verkkoliikennetoiminnan sekä sähköisen asioinnin ratkaisuihin, prosessi-integraatioon ja laadunvarmistukseen. HiQ on perustettu vuonna 1995 ja on listattuna Tukholman pörssissä keskisuurena yhtiönä. HiQ:n liikevaihto oli 1787,9 miljoonaa Ruotsin kruunua vuonna 2017. (HiQ 2018.)

Työn tavoitteena on toteuttaa ja testata FaaS-palveluun perustuvaa taustajärjestelmän soveltuvuutta osana aktiivisessa kehityksessä olevan valvontapalvelun arkkitehtuuria. Tavoitteena on myös käyttää pilvilaskenta-alustana Amazon Web Servicesiä. Alustaa käytetään työssä, koska valvontasovellusta isännöitiin kyseisellä alustalla jo entuudestaan. Työssä keskitytään erityisesti Amazon Web Servicesin FaaS-palveluun Lambdaan ja muihin sen toimintaa tukeviin palveluihin.

Kehitystyössä käytetään apuna Serverless Framework -työkaluohjelmistoa. Ohjelmistolla helpotetaan ja nopeutetaan serverless-arkkitehtuurillisien sovellusten kehitystä ja käyttöönottoa. Työkaluohjelmisto valikoitui käyttöön sen suosion ja alustariippumattomuuden johdosta. Lisäksi Amazonin oma vastaava työkaluohjelmisto oli vasta aikaisessa kehitysvaiheessa.

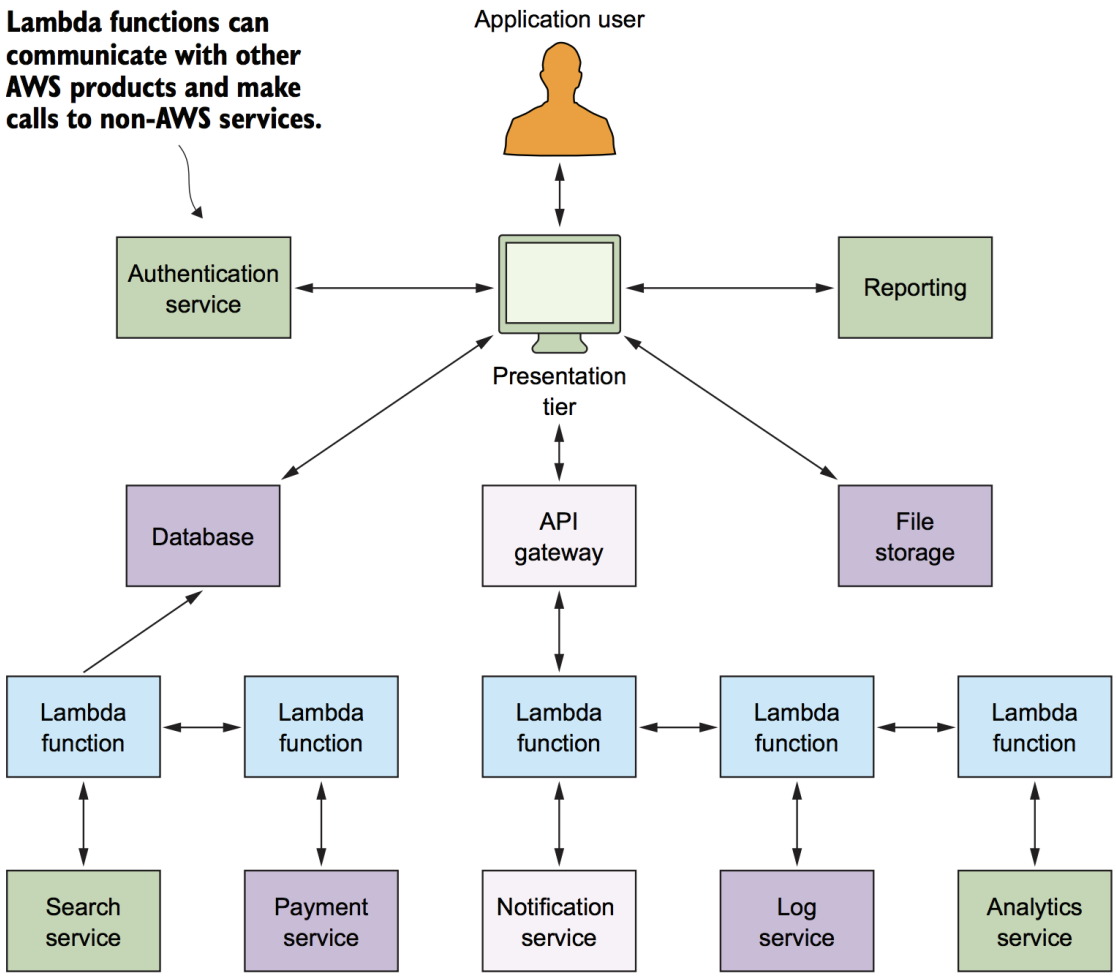
2 SERVERLESS

Perinteisessä web-palvelussa (KUVIO 1) palvelin vastaanottaa ja prosessoi etuosan (front end) lähettämiä http-kutsuja. Data saattaa kulkea useiden sovelluskerrosten läpi ennen sen tallennusta tietokantaan. Taustajärjestelmä generoi vastauksen, joka lähetetään takaisin asiakasohjelmalle (client). (Sbarski 2017, 4.)



KUVIO 1. Perinteinen web-palvelu (Sbarski 2017, 5)

Serverless web-palveluilla (KUVIO 2) ei ole yhtä perinteistä taustajärjestelmää. Sovelluksen etuosa on suorassa yhteydessä palveluiden, tietokannan tai FaaS-funktioiden kanssa API gatewayn välityksellä. Jotkut palvelut on kuitenkin syytä piilottaa FaaS-funktion taakse, jossa voidaan suorittaa täydentäviä turvatoimenpiteitä ja validointeja. (Sbarski 2017, 8.)



KUVIO 2. Serverless web-palvelu (Sbarski 2017, 8)

2.1 Määritelmä

Serverless-palveluilla ei tarkoitetaa sitä, että palvelimista olisi päästy eroon, vaan sitä, että niistä ei tarvitse itse huolehtia. Yleinen teema serverless-palveluissa on juuri se, että palvelinisäntien ja palvelinprosessien hallinnasta ei tarvitse olla enää itse toiminnallisessa vastuussa. Tämä vastuunvapaus on suurin ero muiden ohjelmistojen toimitustapoihin nähden. Serverless kattaa valikoiman tekniikoita ja teknologioita, jotka jaetaan kahteen ryhmään: Backend as a Service (BaaS) ja Functions as a Service (FaaS). (Roberts & Chapin 2017, 6, 39.)

Monet palvelinpuolen ohjelmistojen toimintatavat vaativat sovellusinstanssin käyttöönoton, suorituksen ja monitoroinnin. Perinteisesti sovelluksen elinkaari ulottuu useammalle kuin yhdelle kutsulle; serverless-sovellukset ovat toiminnaltaan päinvastaisia. Serverless-palveluissa ei ole pitkäikäisiä palvelinprosesseja tai palvelinisäntiä hallittavana. FaaS-palveluiden parissa ollaan vielä osallisena ohjelmakoodin käyttöönotossa, mutta tekninen

monitorointi tapahtuu kutsukohtaisesti tai aggregoimalla useiden kutsujen mittareita. (Roberts & Chapin 2017, 40.)

Serverless-palvelut skaalautuvat automaattisesti käyttöasteen mukaan. Skaalaus tapahtuu itsenäisesti ensimmäisestä käyttökerrasta. Automaattiseen skaalaukseen kuuluu myös automaattinen kapasiteetin allokointi, jonka ansiosta päästään eroon suuresta toiminnallisesta vastuusta. (Roberts & Chapin 2017, 40.)

Serveless-palveluiden kustannukset ovat kapasiteetin sijaan tarkasti sidonnaisia käyttöön. Kustannukset voivat vaihdella nollasta ylös ja taas takaisin nollaan. Hinta muodostuu pääosin suoritusajasta. Jos sovellusta suoritetaan muutama minuutti joka tunti, maksetaan jokaiselta tunnilta muutamasta minuutista. BaaS-tietokantojen osalta hinta yleensä muodostuu käytetystä tallennustilasta ja kutsujen määrästä. (Roberts & Chapin 2017, 41.)

Serverless-alustoissa palveluiden suorituskyky määritellään erilaisilla tavoilla, jotka ovat irrallaan pinnan alla olevista instansseista ja isännistä. Alusta määrittelee itse, miten esimerkiksi FaaS-funktiot jaotellaan eri koneille. Funktiot voivat olla sijoitettuna usealle heikommalle koneelle tai vaikkapa yhdelle tehokkaalle koneelle. (Roberts & Chapin 2017, 41.)

Serverless-palveluilla on implisiittinen korkea saatavuus (high availability). Serverless-palveluiden osalta toimittajalta odotetaan läpinäkyvyyttä korkean saatavuuden suhteen. BaaS-tietokantojen osalta oletetaan, että palvelun tarjoaja tekee kaiken mahdollisen yksittäisten isäntien tai sisäisten komponenttien virheiden hallinnan eteen. FaaS:a käytettäessä taas oletetaan, että palvelun tarjoaja uudelleenohjaa kutsuja uuteen funktiokonttiin, jos alkuperäinen ei ole saatavilla. (Roberts & Chapin 2017, 41 - 42.)

2.1.1 Backend as a Service

BaaS:n ymmärtämiseksi käsitteenä on hyvä tietää sen erot muista sitä edeltäneistä pilvilaskennan muodoista, kuten Platform as a Servicesistä (PaaS) ja Software as a Servicesistä (SaaS). PaaS tarjoaa kehittäjille alustan ja ympäristön sovellusten luomiseen. SaaS taas on malli, jossa ohjelmisto lisensoidaan ja toimitetaan. SaaS on osittain BaaS:n kaltainen, mutta kehittäjien sijaan niiden käyttö on suunnattu loppukäyttäjille. (Spoiala 2015.)

BaaS-palveluiden tavoitteena on korvata itse ohjelmoitavia ja ylläpidettäviä palvelinpuolen komponentteja valmiilla palveluilla. Lisäksi palvelut tarjoavat myös rajapinnan ja työkaluja integrointiin eri kielillä. (Spoiala 2015.) BaaS-palvelut ovat kasvattaneet suosiotaan mobiilikehityksessä ja yhden sivun sovelluksien (single page app) kehityksessä, koska niitä

voidaan käyttää suorittamaan tehtäviä, joita muuten oltaisiin itse jouduttu tekemään (Roberts & Chapin 2017, 6).

BaaS-palvelut mahdollistavat turvautumisen toisten implementoimaan ohjelmalogiikkaan. Hyvä esimerkki monissa sovelluksissa toistuvasta ohjelmalogiikasta on autentikointi. Monet sovellukset implementoivat oman koodin muun muassa rekisteröitymiselle, kirjautumiselle ja salasanojen hallinnalle. Useasti tämä koodi on hyvin samankaltaista sovelluksesta toiseen. Tämä onkin johtanut muutamien palveluiden, kuten Auth0:n ja Amazon Cogniton syntymiseen. (Roberts & Chapin 2017, 7.)

2.1.2 Functions as a Service

Serverless-teknologioiden toinen puolisko on Functions as a Service. FaaS on toinen muoto Compute as a Servicestä, geneerisestä ympäristöstä, jossa voidaan suorittaa ohjelmistoja. FaaS on uusi tapa luoda ja ottaa käyttöön palvelinpuolen ohjelmistoja. FaaS on orientoitunut yksittäisten funktioiden tai operaatioiden käyttöönottoon. (Roberts & Chapin 2017, 7.)

FaaS eroaa olennaisesti perinteisestä palvelinpuolen ohjelmiston käyttöönotosta. Perinteisesti ohjelmiston käyttöönotto aloitetaan isäntäkoneesta, joko virtuaalikoneesta tai kontista, jonka jälkeen sovellus otetaan käyttöön isäntäkoneessa. Jos isäntäkone on virtuaalitietokone tai kontti, on sovellus käyttöjärjestelmän prosessi. Yleensä sovellus sisältää ohjelmakoodia useille erillisille toisiinsa liittyville operaatioille. FaaS muuttaa käyttöönottomallia poistamalla siitä isäntäkoneen ja sovellusprosessin. Muunnetussa mallissa keskitytään vain yksittäisiin operaatioihin tai funktioihin, jotka yhdessä muodostavat sovelluksen logiikan. Funktiot lähetetään yksittäin toimittajan FaaS-alustaan. (Roberts & Chapin 2017, 8.)

Alustalle lähetetyt funktiot eivät ole jatkuvasti aktiivisena palvelimen prosessissa odottamassa suoritusta, kuten perinteisessä järjestelmässä. FaaS-alusta on konfiguroitu kuuntelemaan operaatiokohtaisesti tiettyä tapahtumaa, jonka syntyessä funktiota kutsutaan kyseisellä tapahtumalla. Funktion suorituksen päätyttyä, alusta voi tuhota käynnistetyn funktiokontin. (Roberts & Chapin 2017, 9.)

2.1.3 Kontti

Kontit tarjoavat loogisen paketointimekanismin, jossa sovellukset voidaan eristää niiden ajoympäristöstä. Eristys mahdollistaa konttipohjaisten sovelluksien helpon ja luotettavan käyttöönoton kohdeympäristöstä huolimatta. Kontit mahdollistavat sovellusten paketoinnin

kirjastojen ja riippuvuuksien kanssa, tarjoten eristettyjä ympäristöjä ohjelmistopalveluiden suoritukseen. (Google, Inc. 2018.)

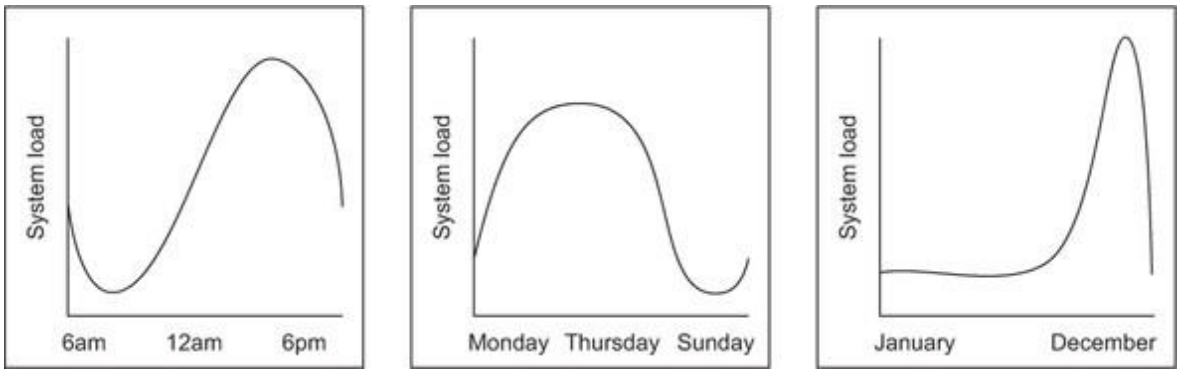
Myös FaaS-palvelut pohjautuvat konttiteknologioihin. Kyseisissä palveluissa konttien hallinnasta ei olla itse vastuussa. Palveluiden funktioita suoritetaan geneerisissä konteissa, joissa on hyvin vähän yleisrasitetta (overhead). (Churchman 2017.)

2.2 Edut

Serverless-palveluilla saavutetaan pienemmät ylläpitokustannukset toimintatyön tarpeen vähenemisellä, sillä muun muassa käyttöjärjestelmien ja tietokantojen versiopäivityksistä ei olla enää itse vastuussa. BaaS-palveluiden käytöstä saatavat hyödyt ovat vielä selkeämmin määriteltävissä, sillä kirjoitetun logiikan määrä vähenee. BaaS-palvelun kattamaa toiminnallisuutta ei tarvitse määritellä, kehittää, testata, käyttöönottaa tai operoida. BaaS-palvelu vaatii kuitenkin integroinnin omaan sovellukseen. FaaS:llä on myös huomattavia kustannuksiin liittyviä etuja perinteiseen lähestymistapaan verrattuna. Ohjelmistokehitys yksinkertaistuu, koska suuri osa infrastruktuurillisesta koodista siirretään alustalle. Lisäksi käyttöönotto tapahtuu helpommin, sillä palveluun lähetetään käytettäväksi vain koodipaketit. (Roberts & Chapin 2017, 20.)

Serverless-palveluiden käyttö pienentää riskiä vähentämällä komponenttien seisonta-aikaa ja niiden pienemmissä määrin vaihtelevaa korjausaikaa. Vaikka virheille ollaan edelleen alttiita sovelluksen eri elementeissä, niistä aiheutuvien riskien hallintatapa on erilainen. Mahdollisten virheidenkorjaustilanteissa voidaan turvautua toisten ammattitaitoon, sen sijaan että ne kaikki korjattaisiin itse. Samalla vähennetään huomattavasti teknologioiden lukumäärää, joista ollaan suorassa operaationaalisessa vastuussa. (Roberts & Chapin 2017, 21 - 22.)

FaaS-palveluista maksetaan suoritusajasta, mikä tarkoittaa, että hinnoittelu skaalautuu käytön mukaisesti. Skaalautuvan hinnoittelun edut ovat huomattavissa erityisesti silloin tällöin tulevien ja vaihtelevien kutsumäärien tapauksissa. Jos palvelun verkkoliikenteessä tapahtuu ajoittain piikkejä tai liikenteen määrä vaihtelee, voi perinteisessä ympäristössä joutua lisäämään laitteiston kapasiteettia kymmenkertaiseksi normaaliin nähden järjestelmän kuormituksen johdosta. Kapasiteetin tarve voi vaihdella huomattavasti ajallisesti (KUVIO 3). Piikkien mukaisesti mitoitettu kapasiteetti taas on ylimitoitettu normaaliin liikenteeseen tai tilanteisiin, joissa liikennettä ei ole laisinkaan. Palvelininstanssien automaattinen skaalaus taas voi olla huono ratkaisu ottaen huomioon palvelimien käynnistysajan, jonka aikana piikki kutsuissa on voinut jo päättyä. Ylimitoitettu kapasiteetti johtaa korkeampaan hintaan, jolta vältytään FaaS-palvelulla. (Roberts 2018.)



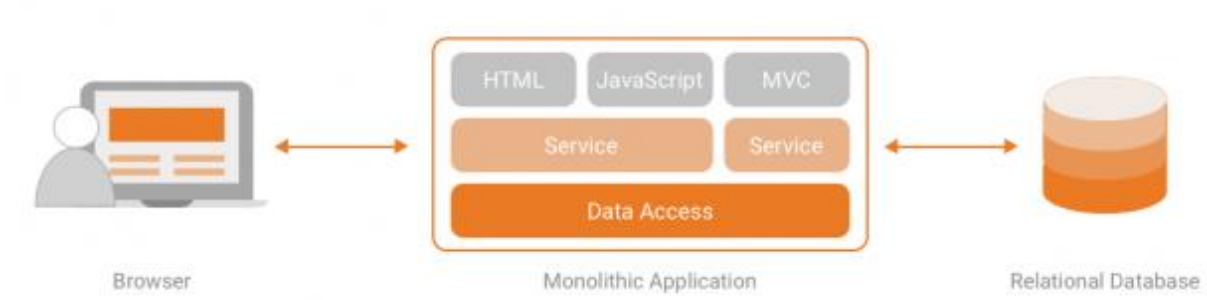
KUVIO 3. Esimerkkejä vaihtelevasta järjestelmän kuormituksesta (Wittig & Wittig 2016, 11)

Käyttämällä serverless-arkkitehtuuria voidaan eliminoida paljon satunnaista monimutkaisuutta tuotannossa olevien sovelluksien tuottamisesta, käyttöönotosta ja operoinnista. Suuressa mittakaavassa näiden monimutkaisuuksien poisto vaikuttaa valtavissa määrin ohjelmistojen toimitukseen. Parhaimmissa tapauksissa voidaan säästää jopa kuukausia kehitysajasta projektissa. (Roberts & Chapin 2017, 26.)

2.3 Erot monoliittisesta arkkitehtuurista

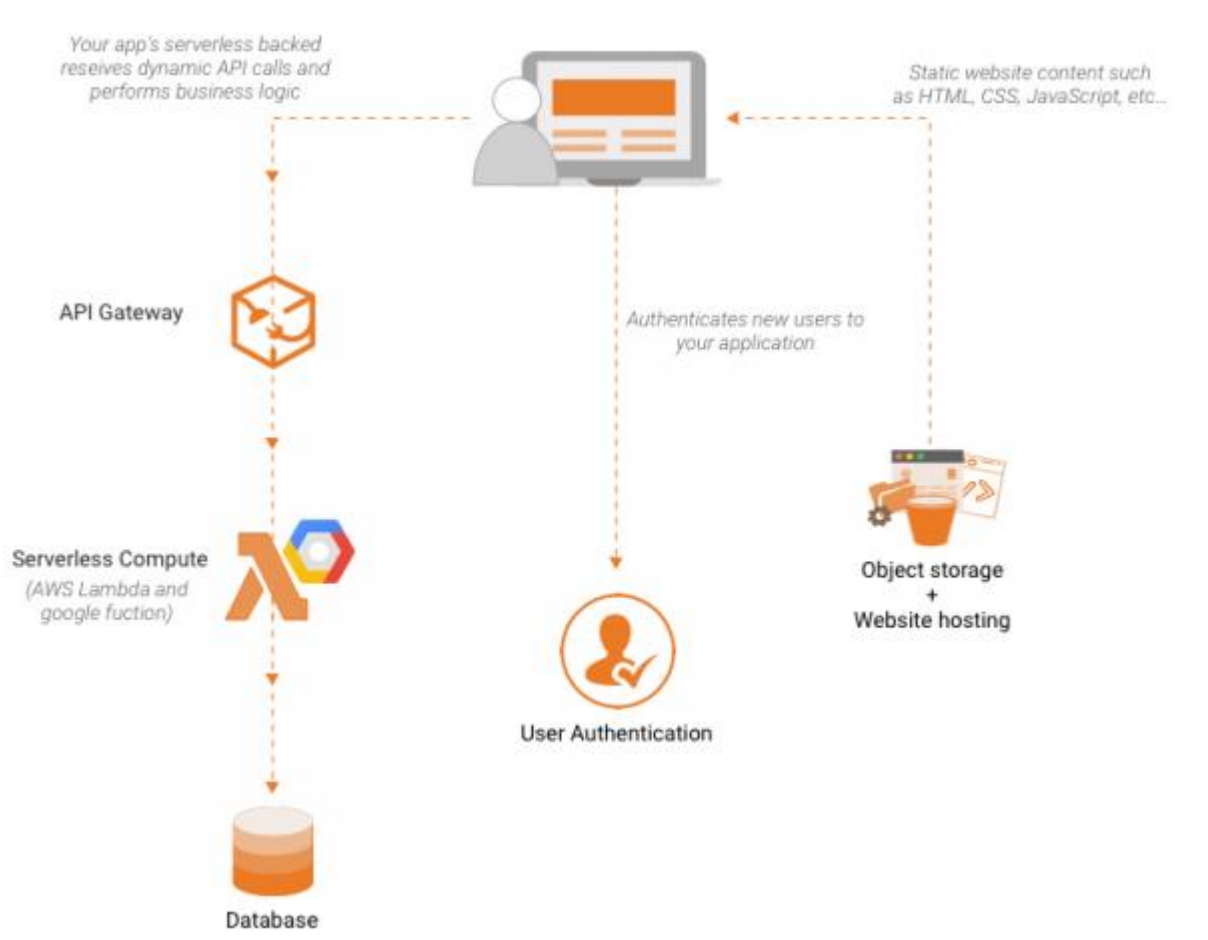
Monoliittisessa arkkitehtuurissa yksi ohjelma tekee kaiken (KUVIO 4). Se esittää käyttöliittymän, käsittelee dataa ja suorittaa sovelluksen vaatimia operaatioita. Monoliittisella arkkitehtuurilla on huomattavia haittapuolia. Erityisesti kaikki järjestelmän osat ovat solmittuna tiukasti yhteen, mikä puolestaan vähentää joustavuutta. (Stephens 2015, 94.)

Kuvion 4 monoliittisessa arkkitehtuurissa sovellus muodostuu kolmesta kerroksesta. Ylin kerros muodostaa sovelluksen käyttöliittymän. Keskimmäinen kerros sisältää liiketoimintalogiikan ja alin kerros on vuorovaikutuksessa tietokannan kanssa. Autentikoinnin ja käyttäjienhallinnan toteutus tässä arkkitehtuurissa vaatii kehitystyötä kaikissa sovellusta muodostavissa kerroksissa.



KUVIO 4. Esimerkki monoliittisesta arkkitehtuurista (Solanki 2018)

Kuvion 4 arkkitehtuurin serverless-toteutuksessa (KUVIO 5) käyttäjien autentikointi ja hallinta hoituu kolmannen osapuolen BaaS-palvelulla. API Gatewayn avulla voidaan reitittää asiakasohjelman ja palvelinpuolen väliset http-kutsut turvallisesti ja skaalautuvasti. Jokainen operaatio on kapseloituna omaan funktioonsa FaaS-palvelussa. FaaS-funktiot voivat integraatioiden ansiosta olla saumattomassa vuorovaikutuksessa BaaS-tietokannan kanssa. Suuri ero monoliittiseen toteutukseen on palvelinpuolen sovelluksen tilattomuus. (Solanki 2018.)



KUVIO 5. Esimerkki serverless-arkkitehtuurista (Solanki 2018)

3 AWS SERVERLESS PLATFORM

Amazon Web Services (AWS) tarjoaa joukon täysin hallittuja palveluita, joita voidaan käyttää serverless-sovellusten luomiseen ja suorittamiseen (Amazon Web Services 2018g). Tarjottuja palveluita käytetään useasti yhdessä, jolloin aikaansaadaan toiminnallisia kokonaisuuksia. Sovelluksen tallennustilana voidaan käyttää Amazon Simple Storage Service -palvelua (Amazon S3): rajapintojen luontiin, julkaisuun, ylläpitoon, monitorointiin ja suojaukseen Amazon API Gateway -palvelua; tietokantana Amazon DynamoDB -palvelua ja laskentaan AWS Lambda -palvelua (Amazon Web Services 2018g).

3.1 Lambda

Lambda on AWS:n tarjoama FaaS-palvelu. Lambda tarjoaa pilven logiikkatason sovellukselle. Lambda-funktioita voidaan laukaista erilaisilla tapahtumilla, joita ilmenee AWS-pilvilaskenta-alustassa tai muissa tuetuissa kolmannen osapuolen palveluissa. Lambdat mahdollistavat reaktiivisten, tapahtumapohjaisten järjestelmien luomisen. Useiden yhdenaikaisten tapahtumien tapauksissa Lambda suorittaa useita kopioita funktiosta rinnakkain. Lambda-funktiot skaalautuvat tarkalleen työmäärän mukaisesti, yksittäisen kutsun tasolle asti. Skaalauksen ansiosta tilanne, jossa palvelin tai kontti on toimettomana, on erittäin epätodennäköinen. Arkkitehtuurit, jotka hyödyntävät Lambdoja, ovat suunniteltuja vähentämään tuhlattua kapasiteettia. (Amazon Web Services 2017, 2.)

Lambda ei rajoita ohjelmointia tiettyyn ohjelmistokehykseen tai kirjastoon. Lambdan tukemien ohjelmointikielten listaan kuuluvat JavaScript (Node.js), Python, Java, C# ja Go (Amazon Web Services 2018f, 1). Lambdat tekevät eri ohjelmointikielien sekoituksesta ja yhdistämisestä sovelluskokonaisuuksissa helppoa, koska yksittäisten Lambdojen koodit voidaan kirjoittaa millä tahansa tuetuista kielistä. On myös mahdollista suorittaa tukemattomien kielien koodia kutsumalla sitä tuetun kielen kautta (Amazon Web Services 2017, 5).

Jokainen luotu Lambda-funktio sisältää suoritettavan ohjelmakoodin, konfiguraation ja vaihtoehtoisesti yhden tai useamman funktion laukaisevan tapahtumalähteen. Konfiguraatiossa määritellään, miten funktion koodi suoritetaan. Koodissa voidaan suorittaa mitä tahansa liiketoimintalogiikkaa, ottaa yhteyttä ulkopuolisiin verkkopalveluihin, integroida muiden AWS-palveluiden kanssa tai tehdä jotain muuta sovelluksen vaatimaa. Kaikki valitun kielen ominaisuudet ja ohjelmistosuunnittelun periaatteet pätevät myös Lambdaa käytettäessä. (Amazon Web Services 2017, 3 - 4.)

Lambdan funktiokoodipaketti sisältää kaikki assetit, jotka halutaan olevan käytettävissä koodia suoritettaessa. Paketin tulee vähintään sisältää funktion koodin, joka halutaan suorittaa kutsusta. Muut assetit voivat olla esimerkiksi tiedostoja ja kirjastoja, joita tuodaan käytettäväksi funktiossa. Funktiokoodipaketin maksimikoko on pakattuna 50 megatavua ja purettuna 250 megatavua. Funktiokoodipaketit ladataan Amazonin S3-buckettiin, josta ne ladataan suoritettavaksi Lambdaan. Bucketit ovat S3:n säiliötä, joihin voidaan tallettaa erilaisia objekteja. (Amazon Web Services 2017, 5.)

Versioinnin avulla on mahdollista julkaista yksi tai useampi versio Lambda-funktiosta. Versiointia voidaan käyttää eri Lambda-funktioiden variaatioiden käyttöön eri ympäristöissä, kuten kehitys- ja tuotantoympäristöissä. Jokaisella Lambdan versiolla on uniikki Amazon Resourse Name (ARN), joka on version julkaisun jälkeen muuttumaton. ARN on yksiselitteinen tapa määritellä resurssi AWS-pilvilaskenta-alustassa, ja se muodostuu kuvion 6 syntaksin mukaisesti. Kuviossa 7 esitetään käyttöönotetun Lambdan ARN. Lambdat tukevat myös peitenimiä (alias) eri versioille. Peitenimi toimii osoittimena tiettyyn funktion versioon, mutta sillä on version tapaan myös oma ARN. Peitenimet ovat muutettavissa, eli niitä voidaan päivittää osoittamaan toisiin versioihin. (Amazon Web Services 2018f, 277 - 278.)

```
arn:aws:lambda:region:account-id:function:function-name
arn:aws:lambda:region:account-id:function:function-name:alias-name
arn:aws:lambda:region:account-id:function:function-name:version
arn:aws:lambda:region:account-id:event-source-mappings:event-source-mapping-id
```

KUVIO 6. AWS Lambdan ARN-syntaksi (Amazon Web Services 2018d, 145)

ARN - arn:aws:lambda:eu-west-1:685198505617:function:serverless-dev-kpis

KUVIO 7. Käyttöönotetun Lambdan ARN

Lambdan koodin suoritus aloitetaan käsittelijästä, joka on tietty metodi tai funktio paketissa. Käsittelijä määritellään Lambda-funktiota kirjoitettaessa. Tuetuilla kielillä on omat vaatimukset siitä, miten käsittelijäfunktio voidaan määritellä (KUVIO 8). Kun käsittelijää on kutsuttu onnistuneesti, voidaan koodissa suorittaa haluttua logiikkaa. Logiikka kannattaa eristää käsittelijäfunktiosta kutsumalla käsittelijässä toista funktiota sen saamilla parametreilla. Toinen funktio voi olla määriteltynä myös toisessa tiedostossa. Logiikan eristäminen käsittelijästä parantaa testattavuutta ja mahdollistaa koodin uudelleenkäytön muualla kuin Lambdoissa. (Amazon Web Services 2017, 6 - 7, 36.)

```
1   exports.handler = function(event, context, callback) {}
```

KUVIO 8. Lambdan käsittelijäfunktio kirjoitettuna JavaScript-kielellä

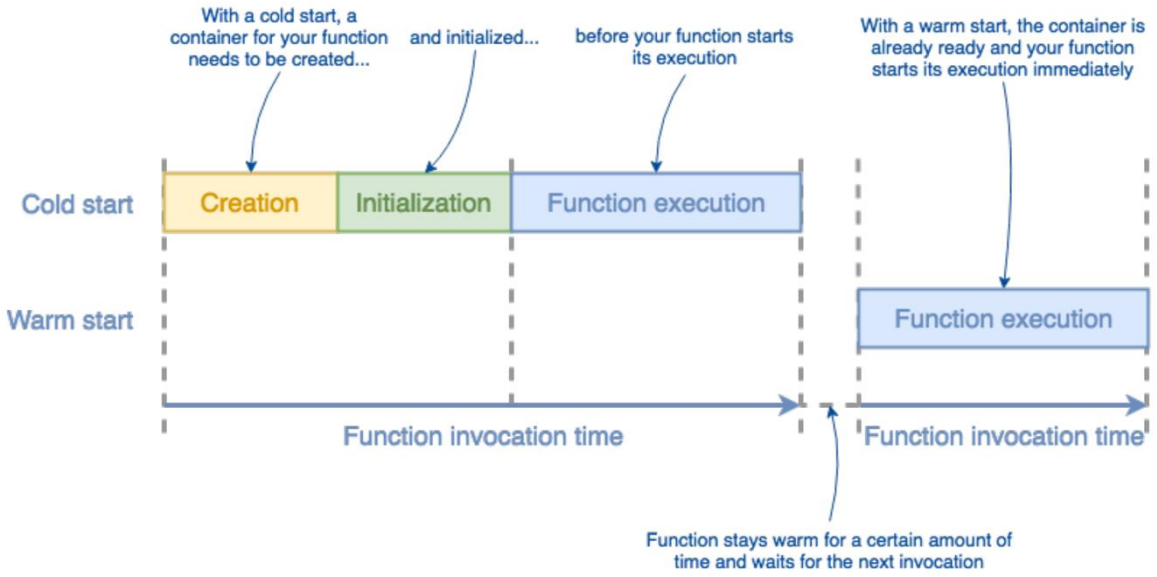
Lambda-funktiota kutsuttaessa käsittelijäfunktio saa parametreina tapahtuma- ja kontekstiobjektit. Tapahtumaobjektin sisältö vaihtelee tapahtuman luojan mukaisesti. API Gatewayn luoma tapatuma sisältää tietoja http-kutsusta ja S3:n uuden objektin luonnista alkunsa saanut tapahtuma sisältää tietoja bucketista ja uudesta objektista. Kontekstiobjekti taas sisältää kielestä riippuen tietoja ajoympäristöstä. Kaikissa kielissä kontekstiobjekti kuitenkin sisältää tiedon kutsutunnisteesta, jäljellä olevasta suoritusajasta ja lokien kohteena olevasta CloudWatch-loki vitauksesta (stream). (Amazon Web Services 2017, 7 - 8.)

Lambda-funktioita voi kutsua kahdella eri mallilla työntömallilla tai vetomallilla. Työntömallin kutsut tapahtuvat toisten AWS-palveluiden tapahtumien kautta. Vetomallin kutsut taas syntyvät, kun Lambdan ajoittaisesti tarkkailema tietolähde vastaanottaa uuden tallenteen. Lambda-funktio voidaan suorittaa synkronisesti tai asynkronisesti. Suoritustapa annetaan InvocationType-parametrina Lambda-funktiota kutsuttaessa. (Amazon Web Services 2017, 10.)

Lambdan suoritusajalle voidaan asettaa aikaraja, jonka ylittyessä tapahtuu aikakatkaisu. Maksimi aikaraja on 300 sekuntia, ja yksittäisen kutsun suoritusaika ei voi ylittää asetettua aikarajaa. Aikakatkaisu ja suorituksen päättyminen laukaisee vastauksen ja kaikki Lambdan käynnistämät taustaprosessit, aliprosessit tai asynkroniset prosessit keskeytetään. Lambdan koodin ei tulisi olla riippuvainen taustaprosesseista tai asynkronisista prosesseista kriittisissä toiminnoissa. (Amazon Web Services 2017, 20 - 21.)

Koodia kirjoittaessa ei voi olettaa Lambdan tilalta mitään. Lambda määrittelee itse, milloin uusi funktiokontti luodaan ja sitä kutsutaan ensimmäistä kertaa. Uudella kontilla on oma tilansa. Uusia kontteja luodaan esimerkiksi, kun käynnissä olevat kontit eivät pysty käsittelemään useita yhdenaikaisia Lambdoja laukaisevia tapahtumia tai jos kontteja ei ole käynnissä kutsun tullessa. Käynnissä olevia Lambdoja kutsutaan lämpimiksi ja uutena luotuja kylmiksi. Lämmin Lambda on aktiivisena muutaman minuutin, kunnes se tuhotaan, jos se ei vastaanota uutta kutsua. Lambdan koodia voidaan optimoida siten, että on mahdollista hyödyntää lämpimiä funktiokontteja. Optimointi voi olla vaikkapa isojen alustettujen objektien tai ulkoisten yhteyksien säilömistä kutsusta toiseen. (Amazon Web Services 2017, 8, 37.) Kylmän ja lämpimän Lambdan kutsuihin kuluvan ajan eroja on havainnollistettu kuviossa 9. Yksinkertaisen ”Hello World” -merkkijonon palauttavalla Lambdalla-funktiolla

kylmän funktion vasteaika voi olla 50-kertainen lämpimään funktioon verrattuna (Malishev 2018).



KUVIO 9. Kylmän Lambdan kutsuun kulutettu aika verrattuna lämpimään (Stojanovic & Simovic 2018, 16)

Rajoittamattoman skaalauksen estämiseksi AWS asettaa oletuksena rajoitteita yhdenaikaisten Lambdojen suoritusten lukumäärään. Oletuksena kaikkien Lambdojen yhdenaikaisten suoritusten yhteenlaskettu lukumäärä ei voi ylittää tuhatta. Rajoitusta voidaan nostaa luomalla tapaus AWS Support Centerissä. (Amazon Web Services 2018f, 390.)

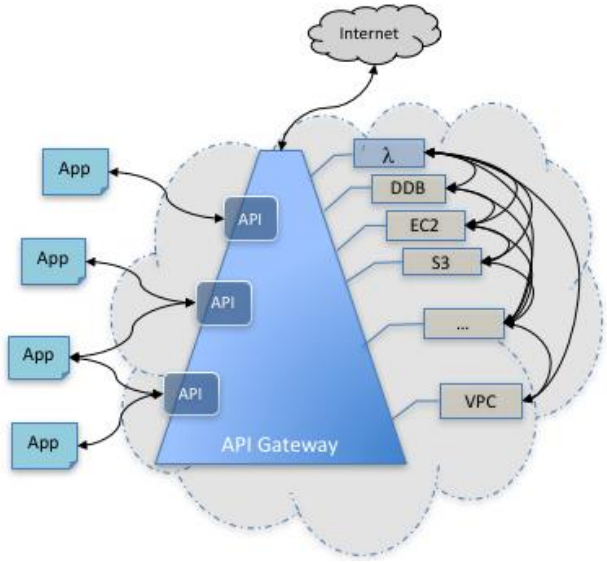
Lambdan hinta perustuu kutsujen määrään sekä suoritusaikaan. Lambda aloittaa suoritusajan laskemisen kutsun vastauksen aloitushetkestä palautukseen tai muuhun keskeytymiseen. Suoritusaikaan lasketaan myös konsolin kautta tehdyt testikutsut. Laskutus tapahtuu kaikkien funktiokutsujen ja suoritusaikojen kokonaismäärän mukaisesti. Suoritusaika pyöristetään lähimpään 100 millisekuntiin. Suoritusajan hinta perustuu funktiolle allokoituun muistimäärään, joka voi olla alimmillaan 128 megatavua ja ylimmillään 3008 megatavua. Lambdalle on saatavilla ilmainen taso, johon kuuluu miljoona kutsua ja 400 000 gigatavusekuntia laskenta-aikaa kuukausittain. (Amazon Web Services 2018e.)

3.2 API Gateway

API Gateway on web- ja mobiilikehittäjille suunnattu palvelu, jonka avulla voidaan tarjota turvallinen ja luotettava tapa päästä käsiksi palvelinpuolen rajapintoihin mobiilisovelluksista, web sovelluksista ja palvelinsovelluksista. API Gateway rajapinnan takana oleva toimintataso voidaan tarjota julkisesti saatavan päätepisteen kautta, jota API Gatewayn

välipalvelimet (proxy) kutsuvat. Vaihtoehtoisesti toimintataso voidaan myös tarjota Lambda-funktioilla. (Amazon Web Services 2018b, 3.)

API Gatewayn arkkitehtuuri (KUVIO 10) mahdollistaa sovelluksille ohjelmallisen pääsyn AWS-palveluihin tai internetin verkkosivuille, yhden tai useamman API Gateway -palvelussa ylläpidetyn rajapinnan kautta. Sovellus on rajapinnan etuosassa; integroidut AWS-palvelut ja verkkosivut ovat rajapinnan takaosassa. API Gateway -palvelussa etuosa on kapseloituna metodikutsuilla ja metodivastauksilla; takaosa on kapseloituna integraatiokutsuilla ja integraatiovastauksilla. Integraatiokutsu on API Gatewayn sisäinen liittymä (interface), joka määrittelee miten metodikutsun parametrit ja sisältö kartoitetaan (map) takaosan vaatimiin muotoihin. (Amazon Web Services 2018b, 2, 5.)



KUVIO 10. API Gatewayn arkkitehtuuri (Amazon Web Services 2018b, 1)

Lambdojen kutsuminen on mahdollista http:n avulla, määrittelemällä mukautettu REST API -päätepiste API Gateway -palvelussa. Yksittäiset metodit, kuten GET ja PUT, voidaan yhdistää omiin Lambda-funktioihinsa. Vaihtoehtoisesti käyttämällä ANY-metodia voidaan yhdistää kaikki tuetut metodit yhteen funktioon. Kun http-kutsu tehdään rajapinnan päätepisteeseen API Gateway -palvelu reitittää sen edelleen Lambda-funktiolle. API Gateway luo kerroksen käyttäjien ja sovelluslogiikan väliin, joka mahdollistaa käyttäjien tai kutsujen järjestelmällisen rajoittamisen, suojauksen palvelunestohyökkäyksiä (DDoS) vastaan ja välimuistikerroksen Lambda-funktioiden vastauksille. API Gateway ei voi kutsua Lambda-funktiota ilman oikeuksia. Oikeudet määritellään Lambda-funktioon liittyvässä lupapolitiikassa (permission policy). (Amazon Web Services 2018f, 146, 161, 234 - 235.)

Lambda proxy-integraatio on yksinkertainen, tehokas ja ketterä mekanismi rajapinnan luomiseen yhden rajapintametodin alkuasetuksella. Proxy-integraatio mahdollistaa

yksittäisen Lambda-funktion kutsumisen asiakasohjelmasta. Proxy-integraation tai mukautetun integraation tarve ilmenee, kun API-kutsut halutaan välittää sellaisenaan asiakasohjelmasta Lambda-funktiolle. Kutsun mukana tulee myös API-konfiguraatiodata, joka voi sisältää esimerkiksi käyttöönottovaiheen nimen (stage name) tai autentikointikontekstin. Proxy-integraation alustus on yksinkertaista. Jos API ei tarvitse sisällön koodausta (encoding) tai välimuistia, on määriteltävä vain integraation http-metodi, integraation päätepisteen URI (Uniform Resource Identifier) kutsuttavan Lambda-funktion ARN:een ja lisätä Identity and Access Management (IAM) -roolille oikeudet Lambda-funktion kutsumiseen API Gatewayn kautta. Mukautettu integraatio vaatii proxy-integraation vaiheiden lisäksi myös saapuvan kutsun datan kartoituksen integraatiokutsuun ja kutsusta syntyvän integraatiovastauksen datan kartoituksen vastausmetodille. Lambda-funktio parsii tulevan kutsun datan määritelläkseen lähetettävän vastauksen. Jotta API Gateway voi välittää Lambdan ulostulon vastauksena asiakasohjelmalle, on funktion palauttaman tuloksen oltava sille yhteensopivassa muodossa (KUVIO 11). Jos vastaus on väärässä muodossa, API Gateway palauttaa vastauksena virheen ”502 Bad Gateway”. (Amazon Web Services 2018b, 132, 140.)

```
{
    "isBase64Encoded": true|false,
    "statusCode": httpStatusCode,
    "headers": { "headerName": "headerValue", ... },
    "body": "..."
}
```

KUVIO 11. API Gatewayn tukema vastausmuoto (Amazon Web Services 2018b, 140)

Proxy-integraatio voidaan asettaa mille tahansa API-metodille, mutta se on toiminnallisesti tehokkaampi konfiguroituna API-metodille, jolla on geneerinen proxy-resurssi. Geneerinen proxy-resurssi voidaan merkitä omalla polkumuuttujalla {proxy+}, ”catch-all” metodilla ANY tai molemmilla. Asiakasohjelma voi välittää syötteen Lambda-funktiolle saapuvassa kutsussa, kutsun parametreina tai käyttökelpoisessa hyötykuormassa (payload). Kutsun parametreihin sisältyvät otsikkotiedot, polkumuuttujat, query-parametrit ja käyttökelpoinen hyötykuorma. Integroitu Lambda-funktio verifioi kaikki syötelähteet ennen kutsun prosessointia ja vastaa asiakasohjelmalle virheviestillä vaaditun syötteen puuttuessa. (Amazon Web Services 2018b, 133.)

Lambdojen tapaan API Gatewayn hinta määräytyy käytön mukaisesti. Minimihintaa tai etukäteen tehtäviä sitoumuksia ei ole. Vain vastaanotetuista rajapintakutsuista ja ulos lähetetystä tiedosta maksetaan. API Gateway tarjoaa myös valinnanvaraisesti välimuistia datalle, josta laskutetaan tuntikohtaisesti riippuen valitusta välimuistin määrästä. API

Gateway voi kuitenkin muodostua toteutetun serverless-sovelluksen huomattavimmaksi kuluksi, Lambda-kutsujen hinnoitteluun verrattuna korkealla hinnoittelulla. Miljoona kutsua maksaa 3,5 dollaria verrattuna Lambdan 0,2 dollariin. (Amazon Web Services 2018a.)

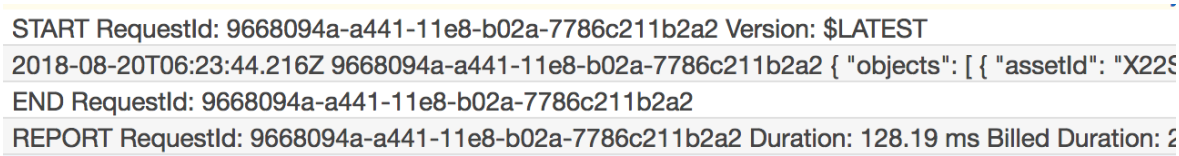
3.3 CloudWatch

Amazon CloudWatch monitoroi reaaliaikaisesti AWS-resursseja ja sovelluksia, jotka ovat suorituksessa AWS-pilvilaskenta-alustalla. CloudWatchia voidaan käyttää mittareiden (metrics) keräykseen ja seurantaan resursseista ja sovelluksista. AWS:n sisäänrakennettujen mittareiden lisäksi on myös mahdollista monitoroida omia mukautettuja mittareita. CloudWatch mahdollistaa järjestelmänlaajuisen näkyvyyden resurssien hyödyntämisestä, sovellusten suorituskyvyistä ja operationaalisista voinneista. CloudWatch-lokeihin pääs-tään käsiksi Amazon CloudWatch -konsolin, AWS CLI:n, CloudWatch API:n ja AWS SDK:n kautta. (Amazon Web Services 2018c, 1.)

Hälytyksiä voidaan käyttää toimintojen aloituksen automatisointiin. Hälytys tarkkailee yksittäistä mittaria määritetyllä ajanjaksolla ja suorittaa yhden tai useamman määritellyn toiminnon, perustuen mittariin suhteutettuna raja-arvoon ajan kuluessa. Hälytykset kutsuvat toimintoja vain jatkuvista tilanmuutoksista eli toimintoja ei kutsuta välittömästi tietyn tilan saavutuksesta, vaan tilan tulee pysyä samana määritetyn ajan. (Amazon Web Services 2018c, 7.)

AWS Lambda monitoroi funktioita automaattisesti ja raportoi niiden mittarit CloudWatchin kautta. Raportoituihin mittareihin lukeutuvat: latenssit, yhteenlaskettu kutsujen lukumäärä ja virhemäärät. Mittareita voidaan hyödyntää mukautettujen hälytysten asettamiseen. (Amazon Web Services 2018f, 330.)

Lambda lokittaa kaikki funktioiden käsittelemät kutsut ja tallettaa myös itse koodissa tehdyt lokitukset CloudWatchiin (KUVIO 12). Lokeja voidaan käyttää myös helpottamaan koodin toiminnallisuuden validointia. Lambdojen CloudWatch-lokit ryhmitellään funktioiden nimen ja etuliitteen ”/aws/lambda/” mukaan. (Amazon Web Services 2018f, 332.)



KUVIO 12. Kuvakaappaus Lambda-funktion CloudWatch-lokeista

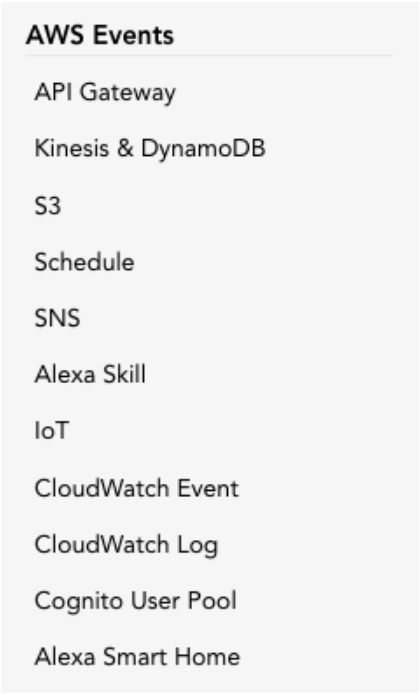
4 SERVERLESS FRAMEWORK

Serverless Framework on työkaluohjelmisto (toolkit), joka tarjoaa perusrakenteita (skaffolding), työnkulun automatisointia ja parhaita käytäntöjä serverless-arkkitehtuuriin pohjautuvien sovelluksien kehitykseen ja käyttöönottoon. Serverless Framework tukee useita eri pilvilaskenta-alustoja, kuten Amazon Web Servicesiä, Microsoft Azurea, IBM OpenWhiskiä ja Google Cloud Platformia. Ohjelmisto on myös laajennettavissa erinäisillä liitännäisillä. Serverless Framework mahdollistaa serverless-mikropalveluiden luonnin ja käyttöönoton minuuteissa. (Serverless, Inc. 2018e.)

Serveless Frameworkin käytöstä on huomattavia etuja, joista yhtenä on lyhyempi kehitysaika. Sen tarjoama komentorivityökalu mahdollistaa kehittäjille serverless-projektien luonnin, testauksen ja käyttöönoton samassa ympäristössä. Kehittäjät määrittävät funktionsa YAML-tiedostoihin, jotka ovat geneerisiä mahdollistaen yhteensopivuuden eri pilvilaskenta-alustojen kanssa. Funktioiden muodostamia palveluita voidaan ottaa käyttöön yhdellä komennolla. Koodin transaktionaalinen käyttöönotto eri alustoissa on myös mahdollista. Lisäksi käyttöönottoja voidaan versioida ja tarvittaessa palauttaa aikaisempaan versioon (roll back). Toisena etuna on yhteen alustaan lukkiutumiselta välttyminen. Kaikilla alustoilla on omat tarvittavat formaatit ja käyttöönottotavat. Serverless Framework kokoaa sovelluksen yhteen pakettiin, joka voidaan ottaa käyttöön eri alustoissa, poistaen pilvikohtaiset muutostarpeet. Kolmantena etuna Serverless Framework integroituu eri serverless-laskentapalveluiden kanssa mahdollistaen infrastruktuurin formalisoinnin ja standardisoinnin koodina. (Serverless, Inc. 2018f.)

Serverless Framework koostuu viidestä ydinkonseptista: funktioista, tapahtumista, resursseista, palveluista ja liitännäisistä. Funktiot ovat esimerkiksi AWS Lambda -funktioita, jotka ovat itsenäisiä käyttöönottoyksiköitä, kuten mikropalvelu. Framework luokittelee tapahtumiksi kaikki funktioita laukaisevat asiat. Eri pilvitoimittajilla on käytettävissä omia palvelukohtaisia tapahtumalähteitä. Kuviossa 13 on esitettynä Serverless Frameworkin tukemat AWS:n tapahtumalähteet. Kun Serverless Frameworkissa määritellään funktioille tapahtuma, Framework luo automaattisesti tarvittavan infrastruktuurin kyseiselle tapahtumalle ja konfiguroi funktion kuuntelemaan sitä. Resurssit ovat pilven infrastruktuurin komponentteja, joita funktiot käyttävät. Serverless Framework käyttöönottaa myös infrastruktuurikomponentit, joista funktiot ovat riippuvaisia. Palvelu on Serverless Frameworkin organisaatioyksikkö. Sitä voidaan pitää projektitiedostona. Yhdessä sovelluksessa voi kuitenkin olla useita palveluita. Palvelussa määritellään funktiot, niitä laukaisevat tapahtumat ja niiden käyttämät resurssit yhdessä serverless.yml-tiedostossa. Serverless Frameworkilla tehdyssä käyttöönotossa, kaikki tiedostossa määritelty otetaan käyttöön kerrallaan.

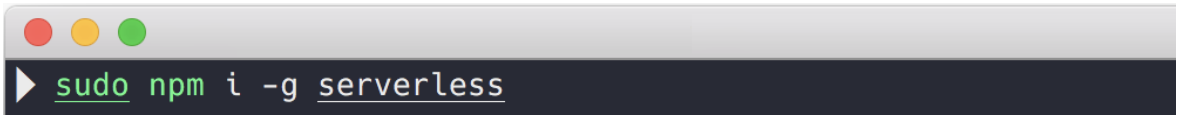
Serverless Frameworkin toiminnallisuutta voidaan ylikirjoittaa ja laajentaa liitännäisillä. (Serverless, Inc. 2018b.)



KUVIO 13. Serverless Frameworkin tukemat AWS-tapahtumalähteet (Serverless, Inc. 2018c)

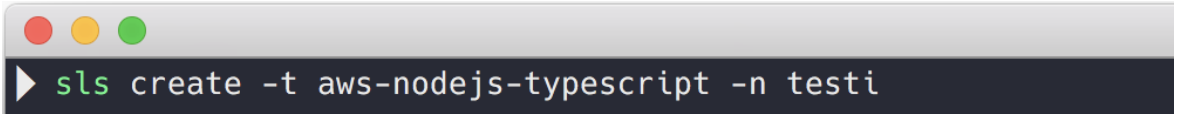
4.1 Asennus ja projektin luonti

Serverless Frameworkin asennus tapahtuu kuvion 14 komennolla. Komento vaatii toimiakseen Node.js:n asennuksen. Komento asentaa työkaluohjelmiston globaalisti, minkä jälkeen se on käytettävissä komentoriviltä. Työkalua voidaan käyttää kirjoittamalla komentoriville ”sls” tai ”serverless”, jota seuraa käytettävä toiminto parametreineen.



KUVIO 14. Komento Serverless Framework -työkaluohjelmiston asennukseen

Asennuksen jälkeen työkalua voidaan käyttää projektin luontiin. Projekti voidaan luoda itse tai käyttämällä sapluunaa (template). Sapluunan käyttäminen nopeuttaa kehityksessä alulle pääsyä huomattavasti. Sapluunoita on tarjolla eri pilvilaskenta-alustoille ja ajoympäristöille. Kuvion 15 komennolla luodaan projekti AWS-pilvilaskenta-alustaa ja Node.js-ajoympäristöä käyttävällä sapluunalla.



KUVIO 15. Esimerkki projektin luonnista ”aws-nodejs-typescript”-sapluunalla

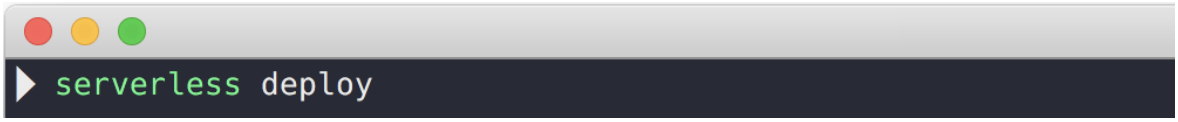
Sapluunan avulla luotu projekti voidaan ottaa käyttöön vasta kun sapluunaa vastaavan pilvilaskenta-alustan valtuustiedot (credentials) ovat asetettuna. Valtuustietojen asetus riippuu pitkälti alustasta. AWS:n tapauksessa kannattaa luoda IAM-käyttäjä oman AWS käyttäjän Identity & Access Management -sivulla, jota sitten käytetään Serverless Frameworkin valtuutukseen. Yksinkertaisinta on luoda käyttäjä ohjelmallisin ylläpito-oikeuksin. Käyttäjän käyttöoikeuksia kuitenkin suositellaan rajoittamaan tuotannossa (Serverless, Inc. 2018a). Käyttäjän luonnin yhteydessä saatu tunniste ja sitä vastaava salainen avainkoodi (secret access key) talletetaan kehitykseen käytettävän tietokoneen ympäristömuuttujiin (KUVIO 16).

```
export AWS_ACCESS_KEY_ID=tunniste
export AWS_SECRET_ACCESS_KEY=salainen-avainkoodi
```

KUVIO 16. Ympäristömuuttujien asetus Unix-järjestelmissä

4.2 Käyttöönotto

Serverless Framework -työkaluohjelmistolla käyttöönotto tehdään päätoimisesti kuvion 17 komennolla. Komento on riippumaton pilvilaskenta-alustasta. Komennon taustalla tapahtuvat toiminnot kuitenkin eroavat toisistaan.



KUVIO 17. Komento Serverless Framework -projektin käyttöönottoon

Kun käyttöönotto tehdään AWS:iin, aluksi serverless.yml-tiedoston sisältö käännetään yhdeksi AWS CloudFormation -sapluunaksi. CloudFormation-sapluuna on tiedosto, jossa määritellään AWS-resursseja, jotka ovat osana CloudFormation-pinoa (stack). Jos pinoa ei ole vielä luotuna, se luodaan ilman resursseja lukuun ottamatta S3-buckettia, jonne funktioiden koodit talletetaan. Kuviossa 18 nähdään pinon luontisapluuna, jossa ainoana resurssina on S3-bucketti. Pinon luonnin jälkeen funktioiden koodit pakataan zip-tiedostoihin. Pakkausta seuraa edellisen käyttöönoton tunnisteiden (hash) vertaus paikallisten tiedostojen tunnisteihin. Käyttöönotto keskeytetään, jos tunnisteet vastaavat toisiaan, muutoin zip-tiedostot ladataan luotuun buckettiin. Latauksen jälkeen IAM-roolit, funktiot ja

resurssit lisätään CloudFormation-sapluunaan ja CloudFormation-pino päivitetään muokatulla sapluunalla. Jokainen käyttöönotto julkaisee uuden version palvelun sisältämistä funktioista. (Serverless, Inc. 2018d.)

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "The AWS CloudFormation template for this Serverless application",
  "Resources": {
    "ServerlessDeploymentBucket": {
      "Type": "AWS::S3::Bucket"
    }
  },
  "Outputs": {
    "ServerlessDeploymentBucketName": {
      "Value": {
        "Ref": "ServerlessDeploymentBucket"
      }
    }
  }
}
```

KUVIO 18. Generoitu sapluuna CloudFormation-pinon luontiin

Koko projektin käyttöönoton sijaan voidaan tehdä myös yksittäisen funktion käyttöönotto. Yksittäisen funktion käyttöönotto ei tee muutoksia CloudFormation-pinoon, vaan se ylikirjoittaa argumenttina annettua nimeä vastaavan funktion Lambda-palvelussa. Tämä tapa on myös paljon nopeampi, koska sen toimintoihin ei käytetä CloudFormationia. (Serverless, Inc. 2018d.)

Kolmas käyttöönottovaihtoehto on käyttöönottaa paketti, joka on valmiiksi luotu kuvion 19 komennolla. Tämä käyttöönotto tapa ohittaa päätoimisen tavan paketointivaiheen ja käyttää olemassa olevaa pakettia käyttöönottoon ja CloudFormation-pinon päivitykseen. (Serverless, Inc. 2018d.)

```
serverless package
```

KUVIO 19. Komento projektin paketointiin

4.3 Funktioiden testaus

Funktioiden testauksella tarkoitetaan tässä yhteydessä Lambda-funktioiden testausta. Funktioiden testaus ei suurelta osin eroa perinteisestä ohjelmakoodin testauksesta. Jos kuitenkin halutaan tehdä integraatiotestejä, vaativat ne valetapahtumia (mock events).

Testauksen helpottamiseksi funktion suorituksen aloittavan käsittelijän tulisi aina olla mahdollisimman riisuttu siinä suoraan suoritettavasta logiikasta ja käyttää hyödykseen moduuleja omista tai ulkopuolisista koodikirjastoista. Jos käytetyt moduulit ovat hyvin yksikkötestattuja, sovelluksen serverless-osuuden eli käsittelijöiden testaus on helppoa integraatiotesteissä. (Hefnawy 2017.)

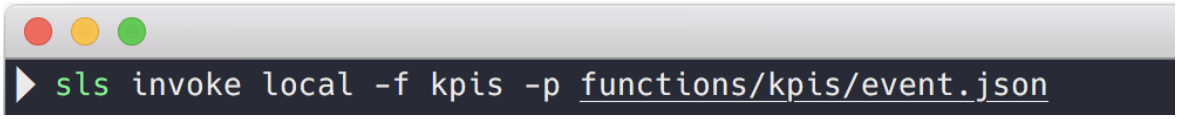
Integraatiotestit vaativat valetapahtumia, jotka vastaavat niiden odottamia tapahtumatyyppejä. Valetapahtumalle voi helposti luoda mallin lokittamalla tapahtuman käyttöönotetussa funktiossa. Lokitettu tapahtuma voidaan kopioida JSON-tiedostoon, joka sitten syötetään funktiolle sitä komentoriviltä kutsuttaessa. (Hefnawy 2017.) Kuviossa 20 on esitettynä lokituksen avulla GET-pyynnöstä luotu valetapahtuma, joka sisältää query-parametreja sekä polkuparametreja.

```
{
  "resource": "/v1/{customerNumber}/kpis",
  "path": "/v1/unmapped/kpis/",
  "httpMethod": "GET",
  "queryStringParameters": {
    "start": "2018-08-20T00:00:00.000Z",
    "end": "2018-08-20T23:59:59.999Z",
    "assetId": "[\"X22SL5651931817\"]"
  },
  "pathParameters": { "customerNumber": "unmapped" },
  "stageVariables": null,
  "requestContext": {
    "resourceId": "4x6rh1",
    "authorizer": {
      "firstName": "John",
      "lastName": "Doe",
      "emailAddress": "john.doe@example.com",
      "premium": "false",
      "customerId": "unmapped",
      "admin": "false",
      "principalId": "unmapped",
      "userName": "user"
    }
  }
}
```

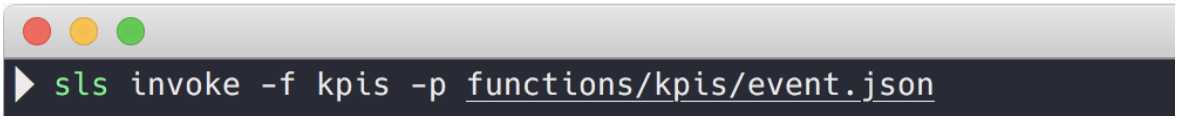
KUVIO 20. Lokitetusta tapahtumasta luotu valetapahtuma

Serverless Frameworkillä funktioita voidaan kutsua kolmella eri tavalla. Paikallisesti funktioita voidaan kutsua kuvion 21 komennolla. Kun funktiota kutsutaan paikallisesti, se

kootaan komennon yhteydessä. Paikallinen funktiokutsu ei emuloi Lambdaa 100 prosenttisesti ja Lambdan rajoitteet eivät päde paikallisesti, joten ennen käyttöönottoa on hyvä tarkistaa, että käytettävissä oleva suoritusaika ja käytetyn muistin suurin sallittu määrä ei ylity. Käyttöönotettuja funktioita voidaan kutsua kuvion 22 komennolla. Käyttöönotettuja funktioita voidaan tietenkin myös kutsua ilman valetapahtumaa, laukaisemalla aito tapahtuma. (Hefnawy 2017.)

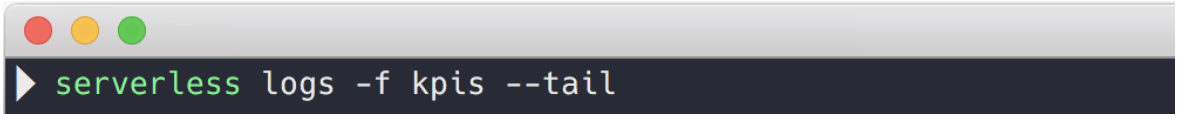


KUVIO 21. Komento funktion kutsumiseen paikallisesti



KUVIO 22. Komento käyttöönotetun funktion kutsumiseen

Funktioiden suorituksen aikaisten virheiden selvittämisen helpottamiseksi funktion käsittelijän sisäinen koodi kannattaa kääriä try/catch-lohkoon. Lambdan lokeja voidaan seurata kuvion 23 komennolla. Suorituksen aikaisia virheitä voi aiheutua myös muista kuin ongelmista koodissa, jolloin try/catch-lohko ei ole riittävä ja ongelman selvitys vaatii lokien tarkastelua. (Hefnawy 2017.)



KUVIO 23. Komento funktion lokien seurantaan

4.4 Yleiset funktioiden arkkitehtuurimallit

Mikropalvelumallilla jokainen tehtävä tai toiminto on eristettynä omaan funktioonsa. Mikropalvelumallin etuna on mahdollisuus muokata sovelluksen komponentteja yksittäin, vaikuttamatta järjestelmään kokonaisuutena. Mikropalvelumallilla luotujen funktioiden määrä on kuitenkin verrattain suuri. (Hefnawy 2016.)

Palvelumallilla yksittäisellä funktiolla on useita tehtäviä, jotka yleensä liittyvät toisiinsa tietorakenteellaan. Tehtäviin voi kuulua esimerkiksi luonti-, luku-, päivitys- ja poisto-operaatiot samalle tietorakenteelle. Palvelumallin funktioiden käyttöönotto on nopeampaa, kuin mikropalvelumallin, sillä luotuja funktioita on vähemmän. Funktio kuitenkin vaatii reitittimen

(KUVIO 24), jonka kautta kutsun metodia tai päätepistettä vastaava logiikka suoritetaan. (Hefnawy 2016.)

```
export const handlerRouterExample: Handler = (
  event: APIGatewayEvent,
  context,
  cb: Callback
) => {
  // Reititys metodin mukaan
  switch (event.httpMethod) {
    case 'GET':
    // ...
  }

  // Reititys polun mukaan
  switch (event.path) {
    case 'v1/example':
    // ...
  }

  // viallinen metodi tai polku
  cb(null, {
    statusCode: 400,
    body: 'Invalid path',
  });
};
```

KUVIO 24. Esimerkki käsittelijässä tapahtuvasta reitityksestä

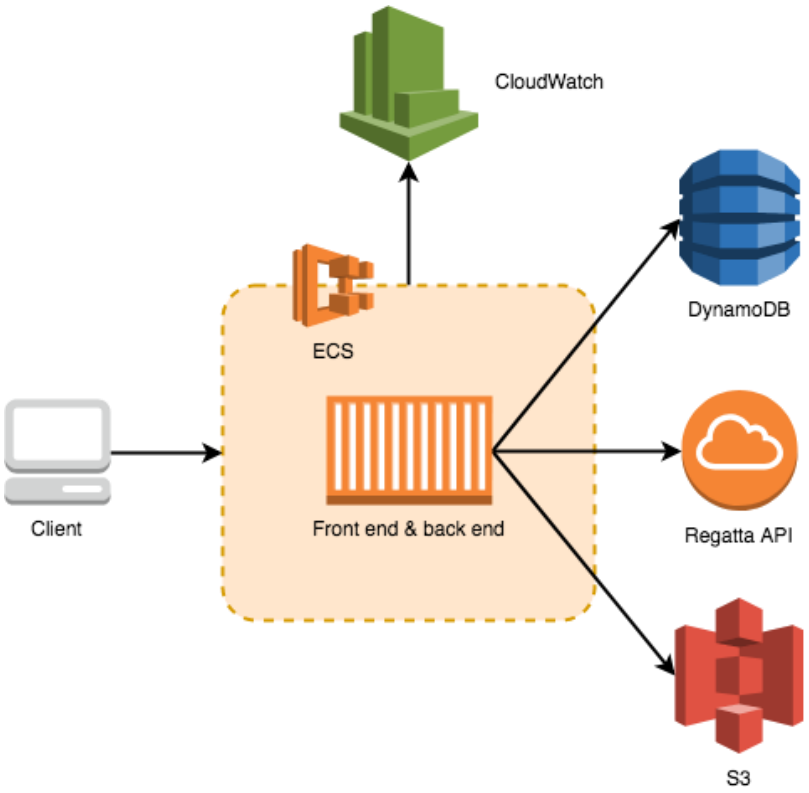
Monoliittimallilla koko sovelluksen toiminnallisuus on yhdessä funktiossa, mikä tarkoittaa, että kaikki päätepisteet viittaavat samaan funktioon. Monoliittimallin funktio saattaa kasvaa helposti kooltaan suurimman sallitun funktiokoon yli. Monoliittimallin funktio vaatii toimiakseen vielä mikropalvelumallin funktion reititintä monimutkaisemman ratkaisun. (Hefnawy 2016.)

5 TAUSTAJÄRJESTELMÄN KEHITYS

Valvontapalvelun käytössä oleva arkkitehtuuri muodostuu suurimmilta osin Amazon Elastic Container Service -palvelussa suoritettavasta kontista. Kontti sisältää sekä sovelluksen etuosan (front end) että taustajärjestelmän. Sovelluskokonaisuus tarjotaan Express-palvelimen kautta. Express on mininaalinen ja joustava Node.js web-sovelluskehys, joka tarjoaa joukon ominaisuuksia web- ja mobiilisovelluksille (Express 2018). Sovelluksen etuosasta tehdään kutsuja taustajärjestelmän tarjoamaan REST-rajapintaan. Tarjotun rajapinnan kautta mahdollistetaan muun muassa sovellukseen kirjautuminen, tiedonhaku tietokannoista ja Regatta-rajapinnasta. Regatta-rajapinnan kautta tarjotaan muun muassa aggregoituja suorituskykymittareita (key performance indicator) laitteille, joilla nostetaan tavaraa.

Express-palvelimelle tehtyjen kutsujen valtuusmerkki (authorization token) tarkistetaan omalla väliohjelmistolla (middleware). Väliohjelmiston avulla voidaan käyttää valtuusmerkin sisältäviä käyttäjätietoja erilaisten operaatioiden suorittamiseen. Käytännössä sovelluksen kaikki sivut ja rajapinnan päätepisteet ovat väliohjelmistolla suojattuja, kirjautumissivua lukuun ottamatta.

Valvontapalvelun käytössä oleva arkkitehtuuri on esitetty kuviossa 25. Kaaviota on kuitenkin yksinkertaistettu jättämällä pois yksittäiset DynamoDB-taulut ja S3-bucketit. Arkkitehtuuriin kuuluu kokonaisuutena myös muita palveluita ja kontteja. Kaaviossa näytetään kuitenkin vain työn kannalta oleelliset osat.

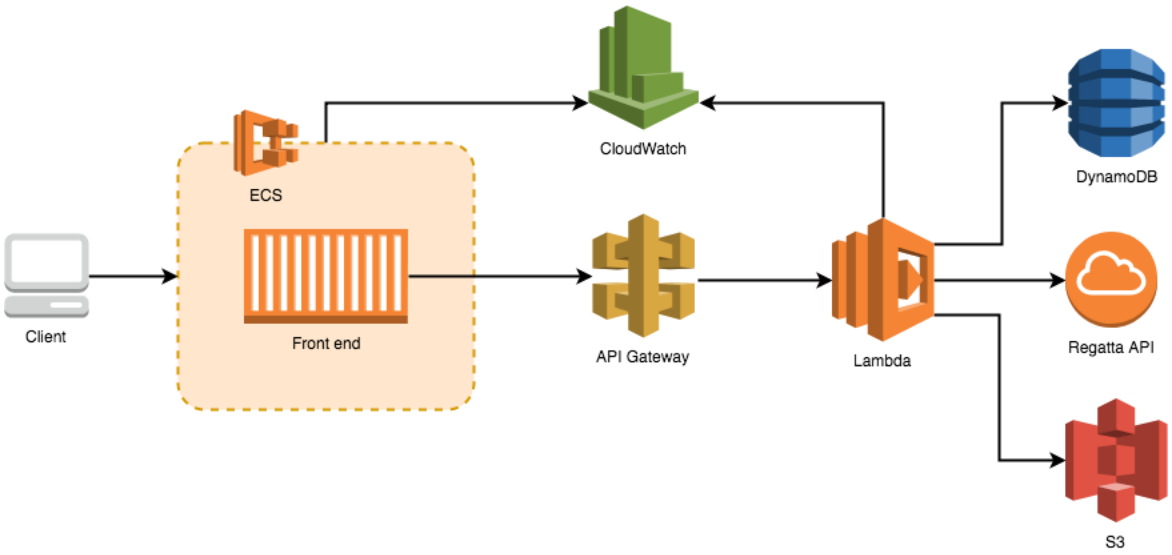


KUVIO 25. Nykyinen käytössä oleva arkkitehtuuri

Valvontapalvelun käyttöliittymä tarjotaan palvelinpuolella mallinnettuna Express-palvelimen avulla. Palvelinpuolen mallinnusta hyödynnetään muun muassa erillisen tulostuspalvelun toiminnan nopeuttamiseksi. Tulostuspalvelulla voidaan nimen mukaisesti tehdä tulostuksia. Tulostukset voivat olla tuettuja käyttöliittymän näkymiä tai automatisoituja raportteja.

5.1 Uusi arkkitehtuuri

Valvontapalvelun uudistettu arkkitehtuuri esitetään kuviossa 26. Uudistetussa arkkitehtuurissa valvontapalvelun taustajärjestelmä tarjotaan REST-rajapinnan muodossa API Gatewayn kautta ja toimintalogiikka toteutetaan Lambda-funktioilla. Lambdat kutsuvat tarpeiden mukaisesti muita AWS-palveluita ja Regatta-rajapintaa. Serverless-arkkitehtuurin kaaviossa ei ole kuvattu jokaista Lambdaa erillisesti, koska kaavion koko paisuisi Lambdojen suuren lukumäärän johdosta.



KUVIO 26. Uusi suunniteltu serverless-arkkitehtuuri

Sovelluksen etuosaa suoritetaan edelleen kontissa, koska palvelinpuolen mallinnuksen toiminnallisuus haluttiin säilyttää. Mallinnus olisi kuitenkin mahdollista toteuttaa myös Lambdassa, mutta suurikokoisten sovelluksien kohdalla Lambda-funktion koko saattaisi olla huomattava. Jos mallinnusta taas ei tarvittaisi, voitaisiin käyttöliittymä tarjota staattisina tiedostoina S3-bucketista.

5.2 Funktioiden arkkitehtuurimallin valinta

Arkkitehtuurimallin valinta vaikuttaa oleellisesti serverless-arkkitehtuurin toteutukseen. Kun toteutuksen toimintalogiikka suoritetaan FaaS-palvelussa, käytetty malli määrittelee tarvittavien funktioiden lukumäärän. Serverless-arkkitehtuuri ei itsessään aseta rajoitteita mallin valintaan. Eri funktiot voivat noudattaa erilaisia malleja.

Taustajärjestelmän toteutukseen funktioiden arkkitehtuurimalliksi valittiin mikropalvelumalli. Suurin osa käytössä olevan taustajärjestelmän rajapinnan päätepisteistä olivat entuudestaan yhden http-metodin päätepisteitä, joka teki niiden muuttamisesta omiksi funktioiksi yksinkertaista. Mikropalvelumallilla vältyttäisiin myös reitittimien kirjoittamisesta aiheutuvalta työltä.

5.3 Projektin luonti

Funktioiden arkkitehtuurimallin valinnan jälkeen voitiin luoda itse projekti. Projektin luontiin käytettiin ”aws-nodejs-typescript”-sapluunaa. Sapluuna sisältää serverless.yml-tiedoston, jossa on määriteltynä pilvilaskenta-alusta, ajoympäristö ja yksi funktio. Sapluunan serverless.yml on täysin muokattavissa. Tiedostossa määritelty ajoympäristö päivitettiin Node.js:n uusimpaan Lambdan käytettävissä olevaan versioon 8.10 ja palvelun nimi

vaihdettiin (KUVIO 27). Provider-ominaisuuden (property) alle määriteltiin myös käytettävä AWS-alue ja funktioiden käyttämiä ympäristömuuttujia.

```
service:
  name: serverless-service
plugins:
  - serverless-webpack
  - serverless-prune-plugin
provider:
  name: aws
  runtime: nodejs8.10
  region: eu-west-1
  environment:
    AUTH_JWT_SECRET:
    AUTH_JWT_ISSUER:
    AUTH_JWT_ALGORITHM:
functions:
  auth:
    handler: functions/auth/auth.auth
```

KUVIO 27. Kuvakaappaus taustajärjestelmän serverless.yml-tiedostosta

5.4 Käytetyt Serverless Framework -liitännäiset

Koska jokainen käyttöönotto luo uudet versiot Serverless Framework -projektin funktioista, on käyttöönotettujen versioiden lukumäärän rajoitukseen kehitetty oma liitännäinen Serverless Prune Plugin. Liitännäisiä voidaan asentaa npm-pakettienhallintaohjelman avulla. Liitännäisellä rajoitettiin projektin käyttöönotettujen versioiden lukumäärä yhteen ja samalla vanhojen versioiden poisto automatisoitiin. Versioiden lukumäärä ja poiston automatisointi määritetään serverless.yml-tiedostossa (KUVIO 28). Poistoprosessi voidaan myös suorittaa kuvion 29 komennolla ilman sitä edeltävää käyttöönottoa.

```
custom:
  prune:
    automatic: true
    number: 1
```

KUVIO 28. Serverless Prune Plugin -liitännäisen asetukset serverless.yml-tiedostossa

```
serverless prune -n 1
```

KUVIO 29. Serverless Prune Pluginin komento, jolla poistetaan funktioita

Toinen projektissa käytetty liitännäinen on Serverless Webpack, jolla mahdollistetaan funktioiden luominen suositun Webpack-moduuliniputtajan (module bundler) avulla. Projektissa Webpackia käytettiin TypeScriptin transpilaukseen JavaScriptiksi. Webpackin konfiguraatio saatiin projektin luonnissa käytetyn sapluunan mukana, mutta sitä jouduttiin kuitenkin muuttamaan funktioiden yksittäistä paketointia tukevaksi. Funktioiden yksittäisellä paketoinnilla aikaansaadaan pienemmät koot käyttöönotetuille funktioille.

5.5 Kutsujen autorisointi

Lambdoille tehtäviä kutsuja voidaan autorisoida määrittelemällä API Gatewayn reiteille Lambda-myöntäjä (authorizer). Serverless Framework -työkaluohjelmistoa käytettäessä myöntäjä määritellään tapahtumakohtaisesti serverless.yml-tiedostossa. Myöntäjä on määriteltynä ”assets”-funktiolle kuviossa 30 rivillä 38.

```
29  functions:
30    auth:
31      handler: functions/auth/auth.auth
32    assets:
33      handler: functions/assets/assets.assets
34      events:
35        - http:
36            method: get
37            path: v1/{customerNumber}/assets
38            authorizer: auth
39            cors: true
```

KUVIO 30. Lambda-myöntäjän määritys serverless.yml-tiedostossa

Lambda-myöntäjä on Lambda-funktio, joka palauttaa generoidun IAM-politiikan (policy). IAM-politiikan lisäksi myöntäjäfunktion on palautettava vastauksessa kutsujan principal id (Amazon Web Services 2018b, 288). Kuvion 31 apufunktiolla voidaan luoda IAM-politiikka, jolla joko sallitaan tai evätään myöntäjällä suojatun API Gateway -päätepistettä vastaavan resurssin kutsuminen. Politiikan mukana voidaan välittää vaihtoehtoisesti ”context”-ominaisuuden sisällä avain-arvo-pareja (Amazon Web Services 2018b, 288). Työssä ”context”-ominaisuuteen asetettiin kutsun valtuusmerkin sisältäneen käyttäjäobjektin avain-arvo-parit kuvion 31 rivillä 22, joita sitten hyödynnettiin Lambdassa tehtävissä kutsuissa.

```
export const generatePolicy = (
  principalId: string,
  effect: string,
  resource: string,
  context?: any
): any => {
  let authResponse: any = {};
  authResponse.principalId = principalId;
  if (effect && resource) {
    let policyDocument: any = {};
    policyDocument.Version = '2012-10-17';
    policyDocument.Statement = [];
    let statementOne: any = {};
    statementOne.Action = 'execute-api:Invoke';
    statementOne.Effect = effect;
    statementOne.Resource = resource;
    policyDocument.Statement[0] = statementOne;
    authResponse.policyDocument = policyDocument;
  }

  if (context) {
    authResponse.context = context;
  }

  return authResponse;
};
```

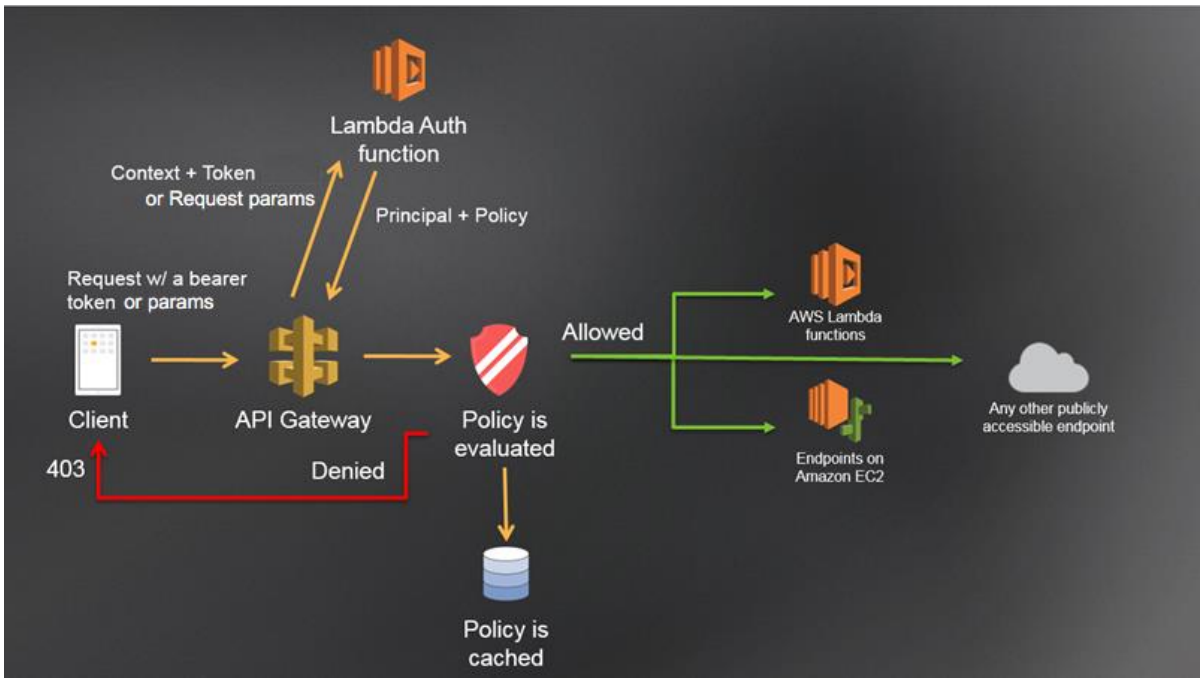
KUVIO 31. Apufunktio IAM-politiikan generointiin

Projektin kutsuille käytetty Lambda-myöntäjä (KUVIO 32) tarkistaa kutsun sisältämän valtuutusmerkin. Valtuusmerkin puuttuessa, sen ollessa väärässä muodossa, vanhentunut tai viallinen, funktio palauttaa viestin ”Unauthorized”, jolloin politiikka on mitätön (invalid), se ei läpäise tarkistusta ja API Gateway -palveluun tehty API-kutsu epäonnistuu. Valtuusmerkin tarkistuksen onnistuessa funktio palauttaa pätevän (valid) IAM-politiikan. Koska samaa myöntäjäfunktiota käytettiin kaikkien funktioiden tapahtumissa, asetettiin politiikan resurssiksi ”*” kuvion 32 rivillä 29. Onnistuneesti evaluoidut myöntäjäfunktiolla autorisoidut politiikat talletetaan välimuistiin (KUVIO 33). Jos välimuistiin talletetun politiikan resurssiksi on asetettuna yksittäisen Lambdan resurssi, päätyvät kaikki muut samalla valtuusmerkillä tehdyt, toisille saman rajapinnan päätepisteille tehdyt kutsut 403 Forbidden -virheeseen. Kun resurssiksi on asetettu ”*” voidaan välimuistiin talletetulla politiikalla kutsua kaikkia muita rajapinnan päätepisteitä. Välimuistille voitaisiin vaihtoehtoisesti asettaa eliniäksi nolla sekuntia, jolloin resurssi voisi olla kutsun tapahtuman mukana saatava methodArn.

Eliniän asetus nollaksi tarkoittaa kuitenkin, että jokaisen funktiokutsun yhteydessä kutsuttaisiin myös myöntäjäfunktiota, joka ei ole kovin kustannustehokas ratkaisu, sillä jokaisesta tehdystä kutsusta maksetaan.

```
import { verify } from 'jsonwebtoken';

import { generatePolicy } from '../../utils';
import { DecodedToken } from '../../types';

export const auth = (event, context, cb) => {
  if (!event.authorizationToken) {
    return cb('Unauthorized');
  }

  const [schema, tokenValue] = event.authorizationToken.split(' ');
  if (!(schema.toLowerCase() === 'bearer' && tokenValue)) {
    return cb('Unauthorized');
  }

  try {
    verify(
      tokenValue,
      process.env.AUTH_JWT_SECRET,
      {
        issuer: process.env.AUTH_JWT_ISSUER,
        algorithms: [process.env.AUTH_JWT_ALGORITHM],
      },
      (verifyError, decoded: DecodedToken) => {
        if (verifyError) {
          console.log(`Token invalid. ${verifyError}`);
          return cb('Unauthorized');
        }
        return cb(null, generatePolicy(decoded.sub, 'Allow', '*', decoded.user));
      }
    );
  } catch (err) {
    console.log('Invalid token', err);
    cb('Unauthorized');
  }

  return cb('Unauthorized');
};
```

KUVIO 32. Lambda-myöntäjän toteutus



KUVIO 33. Myöntäjällä suojatun API Gateway -päätepisteen työnkulku (Amazon Web Services 2018b, 285)

5.6 Lambda-esimerkki

Kuviossa 34 on nähtävissä suorituskykymittaritietoja hakeva Lambda. Suorituskykymittaritiedot haetaan sivun 24 kuviossa 25 esitetystä Regatta-rajapinnasta. Regatta-rajapintaan tehtävistä kutsuista vastaa RegattaDataService-apuluokka. Apuluokkaa hyödynnetään kaikissa samaa rajapintaa kutsuvissa Lambdoissa. Apuluokka sisältää erilaisia metodeja, joiden avulla kutsut suoritetaan. Apuluokka luo kutsun Regatta-rajapinnalle sen metodeille syötettyjen parametrien pohjalta. Kuvion 34 Lambdaa voidaan kutsua kuviota 35 vastaavalla pyynnöllä.

```
1  import moment from 'moment-timezone';
2  import { APIGatewayEvent, Callback, Context } from 'aws-lambda';
3
4  import RegattaDataService from '../../helpers/RegattaDataService';
5  import { isPremium, lambdaProxyResponse } from '../../utils';
6
7  const rds = new RegattaDataService();
8
9  export const kpis = async (
10   event: APIGatewayEvent,
11   context: Context,
12   cb: Callback
13 ) => {
14   try {
15     const premium = isPremium(event.requestContext);
16     const { assetId, start, end } = event.queryStringParameters;
17     const res = await rds.fetchKpis({
18       assetIds: JSON.parse(assetId),
19       timeZone: 'UTC',
20       start: start ? moment(start) : start,
21       end: end ? moment(end) : end,
22       premium,
23     });
24     cb(null, lambdaProxyResponse(res));
25   } catch (e) {
26     console.log(e.message || e);
27     cb(e);
28   }
29 };
```

KUVIO 34. Suorituskykymittaritietojen hausta vastaava Lambda-funktio

```
3  GET /dev/v1/unmapped/kpis
4  ?assetId=["X22SL5651931817"]
5  &start=2018-08-20T00:00:00.000Z
6  &end=2018-08-20T23:59:59.999Z HTTP/1.1
7  Host: {{host}}
8  Authorization: Bearer {{token}}
```

KUVIO 35. Suorituskykymittaritietoja hakevalle Lambdalle tehty kutsu

Lambda-esimerkin käsittelijään suoraan liitetty logiikka on minimoitu ja apuna kutsun käsittelyyn käytetty apufunktioita ja apuluokkaa. Käytetty apuluokka RegattaDataService on alustettuna käsittelijän ulkopuolella kuvion 34 rivillä 7. Alustus käsittelijän ulkopuolella mahdollistaa saman instanssin käytön seuraavissa vastaanotetuissa kutsuissa.

Käsittelijän määrittely alkaa kuviossa 34 riviltä 9 ja kaikki sen ulkopuolinen koodi suoritetaan vain Lambdan käynnistysvaiheessa.

Käsittelijässä käytetty isPremium-funktio tarkistaa Lambda-myöntäjän tapahtumaan lisäämistä käyttäjätiedoista onko käyttäjä premium-käyttäjä. Kutsusta luotuun tapahtumaan sisältyvät query-parametrit ovat saatavissa tapahtuman queryStringParameters-ominaisuudesta. Tapahtumasta muodostetaan parametrit RegattaDataService-luokan metodikutsuun.

RegattaDataServicen fetchKpis-metodi luo parametrien perusteella Regatta-rajapintaan yhteensopivan kutsun. Rajapinnalta vastaanotettu data on kuvion 36 muodossa. Vastaanotettu data ei sovellu sellaisenaan sovelluksen etuosan käytettäväksi, joten sama metodi muuntaa datan käytettävämpään muotoon (KUVIO 37) ennen sen palautusta.


```
{
  "objects": [
    {
      "assetId": "X22SL5651931817",
      "signalNames": [
        "CycleCntFastLoad_hourly_delta",
        "CycleCntFastLowering_hourly_delta",
        "CycleCntFastTipping_hourly_delta",
        "CycleCntFastUnload_hourly_delta",
        "CycleCntLoadUnload_hourly_delta",
        "CycleCntTipping_hourly_delta",
        "CycleCntTotal_hourly_delta",
        "DrivingDistance",
        "DrivingTime",
        "FastSpeedCounter_hourly_delta",
        "HourMeter_hourly_delta",
        "RemoteControlCounter_hourly_delta"
      ],
      "sampleValueRows": [
        {
          "time": "2018-08-20T00:00:00Z",
          "values": [0, 3, 3, 15, 17, 3, 20, 252071, 16156, 12, 29, 1]
        }
      ]
    }
  ]
}
```


KUVIO 36. Regatta-rajapinnalta vastaanotettuja suorituskykymittaritietoja

```
{
  "2018-08-20": [
    {
      "assetId": "X22SL5651931817",
      "time": "2018-08-20",
      "cycleCntFastLoadHourlyDelta": 0,
      "cycleCntFastLoweringHourlyDelta": 3,
      "cycleCntFastTippingHourlyDelta": 3,
      "cycleCntFastUnloadHourlyDelta": 15,
      "cycleCntLoadUnloadHourlyDelta": 17,
      "cycleCntTippingHourlyDelta": 3,
      "cycleCntTotalHourlyDelta": 20,
      "drivingDistance": 252071,
      "drivingDistanceKm": 252.071,
      "drivingTime": 16156,
      "drivingTimeHours": 4.487777777777778,
      "fastSpeedCounterHourlyDelta": 12,
      "hourMeterHourlyDelta": 29,
      "hourMeterHourlyDeltaHours": 0.48333333333333334,
      "remoteControlCounterHourlyDelta": 1
    }
  ]
}
```


KUVIO 37. RegattaDataService-luokan fetchKpis-metodin palauttama data

Serverless Framework luo määritetyt funktiot oletuksena proxy-integraatiolla, joka tarkoittaa, että Lambdan palauttama vastaus on oltava sivun 14 kuviossa 11 esitetyssä muodossa. Vastauksen sisällön on oltava muunnettavissa merkkijonoksi JSON.stringfy() -metodilla. Vastauksen muodostamiseen luotiin oma apufunktio lambdaProxyResponse (KUVIO 38), jota voitiin sitten käyttää kaikissa Lambdoissa. LambdaProxyResponse-funktiota käytetään kuviossa 34 rivillä 24 ja funktion palauttama data on kuvion 39 muodossa.

```
export const lambdaProxyResponse = (response): ProxyResult => ({
  isBase64Encoded: false,
  statusCode: 200,
  headers: {
    'Access-Control-Allow-Headers': '*',
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Credentials': true
  },
  body: JSON.stringify(response),
});
```

KUVIO 38. Apufunktio proxy-integroidun Lambdan vastauksen luontiin

```
{
  "isBase64Encoded": false,
  "statusCode": 200,
  "headers": {
    "Access-Control-Allow-Headers": "*",
    "Access-Control-Allow-Origin": "*",
    "Access-Control-Allow-Credentials": true
  },
  "body": "{\"2018-08-20\":[{\"assetId\":\"X22SL5651931817\",
  \"time\":\"2018-08-20\",\"cycleCntFastLoadHourlyDelta\":0,
  \"cycleCntFastLoweringHourlyDelta\":3,\"cycleCntFastTippingHourlyDelta\":3,
  \"cycleCntFastUnloadHourlyDelta\":15,\"cycleCntLoadUnloadHourlyDelta\":17,
  \"cycleCntTippingHourlyDelta\":3,\"cycleCntTotalHourlyDelta\":20,
  \"drivingDistance\":252071,\"drivingDistanceKm\":252.071,
  \"drivingTime\":16156,\"drivingTimeHours\":4.487777777777778,
  \"fastSpeedCounterHourlyDelta\":12,\"hourMeterHourlyDelta\":29,
  \"hourMeterHourlyDeltaHours\":0.48333333333333334,
  \"remoteControlCounterHourlyDelta\":1}]}"
}
```

KUVIO 39. Lambdalle tehdyn kutsun vastauksena saatu data

5.7 CORS-tuki

CORS-tuella mahdollistetaan kehitetyn taustajärjestelmän kutsumisen eri lähteestä. CORS-tukeen vaaditaan API Gatewayn päätepisteille OPTIONS-metodit ja vastauksiin ylimääräisiä http-otsikkotietoja. Serverless Framework yksinkertaistaa päätepisteen määrittelyn tarjoamalla tapahtuman alle määriteltävissä olevan cors-ominaisuuden. Cors-ominaisuus on määritelty kuviossa 40 rivillä 55. Proxy-integraatiota käyttäville funktioille määritellään vastaus itse funktion koodissa, joten otsikkotiedot voidaan määritellä vaikka onnistuneen vastauksen muodostamiseen käytetyssä apufunktiossa (KUVIO 38). Kuviossa 41 on nähtävissä CORS-tuetun Lambdan vastauksessa saadut otsikkotiedot.

```
48    kpis:
49      handler: functions/kpis/kpis.kpis
50      events:
51        - http:
52            method: get
53            path: v1/{customerNumber}/kpis
54            authorizer: auth
55            cors: true
```

KUVIO 40. Esimerkki CORS-tuetusta funktion tapahtumasta

```
1   HTTP/1.1 200 OK
2   Content-Type: application/json
3   Content-Length: 1101
4   Connection: keep-alive
5   Date: Tue, 21 Aug 2018 06:13:24 GMT
6   x-amzn-RequestId: 4f811c25-a509-11e8-b6da-cd457c6de0b8
7   Access-Control-Allow-Origin: *
8   Access-Control-Allow-Headers: *
9   x-amz-apigw-id: L9gMpG6lDoEFVRw=
10  X-Amzn-Trace-Id: Root=1-5b7bad84-309a1bb6ca4e51c7c84d7ed1;Sampled=0
11  Access-Control-Allow-Credentials: true
12  X-Cache: Miss from cloudfront
13  Via: 1.1 89911e4e5378c3240f7dee234d09f038.cloudfront.net (CloudFront)
14  X-Amz-Cf-Id: tEIRORplc2jlC0FdVYqEB9ts3Uieo0MZGH8IHF6PD2hhhLTR21Rcqg==
```

KUVIO 41. CORS-tuetun Lambdan vastauksen otsikkotiedot

Kun funktioiden tapahtumille on asetettuna Lambda-myöntäjä, vaaditaan CORS-tuen saavuttamiseksi myös erillisen resurssin määrittely. Kuviossa 42 määritelty resurssi on mukautettu vastaus API Gatewaylle. Vastauksen tulee sisältää vaaditut otsikkotiedot Access-Control-Allow-Origin, Access-Control-Allow-Headers ja Access-Control-Allow-Credentials, jotta käyttäjän selain tietää mitä tehdä myöntäjän evätessä kutsun. Oletuksena lähetetty vastaus ei sisällä vaadittuja http-otsikkotietoja.

```
133   resources:
134     Resources:
135       GatewayResponseDefault4XX:
136         Type: 'AWS::ApiGateway::GatewayResponse'
137         Properties:
138           ResponseParameters:
139             gatewayresponse.header.Access-Control-Allow-Origin: "'*'"
140             gatewayresponse.header.Access-Control-Allow-Headers: "'*'"
141             gatewayresponse.header.Access-Control-Allow-Crendentials: "'true'"
142           ResponseType: DEFAULT_4XX
143           RestApiId:
144             Ref: 'ApiGatewayRestApi'
```

KUVIO 42. API Gatewayn vastausresurssin määrittely serverless.yml-tiedostossa

6 YHTEENVETO

Työn tavoitteena oli toteuttaa ja testata FaaS-palveluun pohjautuva taustajärjestelmä osaksi valvontapalvelun arkkitehtuuria. Taustajärjestelmä tarjoaa rajapinnan lukuisten eri operaatioiden suorittamiseen. Saman rajapinnan kautta tehdään myös kutsuja Regatta-rajapintaan.

Koska työn projektissa käytettiin entuudestaan Amazon Web Services pilvilaskenta-alustaa, hyödynnettiin toteutuksessakin samaa alustaa. Uuden projektin aloitusvaiheessa kannattaa kuitenkin tarkastella eri alustojen palvelutarjontaa ja ottaa huomioon käytettävistä palveluista muodostuvat kustannukset kokonaisuutena, eikä vain yksittäisen palvelun hinnan perusteella. Esimerkiksi Lambdoja käyttävää REST -rajapintaa ei voida luoda AWS-pilvilaskenta-alustassa käyttämättä Lambdaa yhdessä API Gateway -palvelun kanssa. API Gatewayn käytöstä maksetaan Lambdan lisäksi. Käytettävissä on myös muita alustoja, joissa erillistä API gateway -palvelua ei tarvita tai vastaava palvelu on ilmainen.

Lambdoihin perustuvan taustajärjestelmän etuna käyttöönotossa konttipohjaiseen ratkaisuun on mahdollisuus tehdä muutoksia yksittäisen funktion toimintaan. Kun muutokset on tehty, ei ole tarpeellista luoda uutta kuvaa kontista, vaan sen sijaan voidaan tehdä käyttöönotto yksittäiselle funktiolle. Yksittäisen Serverless Framework -projektin funktion käyttöönottoa ei kuitenkaan suositella tuotannossa, sillä se vaatii aina olemassa olevan CloudFormation-pinon. Lisäksi yksittäisten funktioiden käyttöönotot ylikirjoitetaan projektin käyttöönoton yhteydessä. Yksittäisen funktion käyttöönotto on kuitenkin nopeampaa kuin kokonaisen projektin ja se voi olla hyvä vaihtoehto kehitysympäristössä nopeaan iterointiin, kunhan toiset kehittäjät eivät tee samanaikaisesti käyttöönottoja kyseisestä funktiosta tai koko projektista.

Itse Lambdojen kirjoittaminen on suoraviivaista. Serverless Frameworkin serverless.yml-tiedoston määrittely ottaa oman aikansa, mutta kun alustava konfiguraatio on luotu, uusien funktioiden lisääminen on helpompaa ja projektin paisuessa, sitä voidaan pilkkoa useisiin projekteihin.

Monissa tilanteissa Lambda-funktion oletusmuistimäärä on ylimitoitettu sen käyttötarpeeseen. Tarvittava muistin määrä kannattaakin selvittää CloudWatch-lokien perusteella sekä paikallisesti funktioita suorittamalla. AWS Lambda -palvelussa Lambda-funktion suorituskyky on sidonnainen sen muistinmäärään. Muistinmäärän sidonnaisuus suorituskykyyn voi johtaa tilanteisiin, joissa on paljon ylimääräistä muistia.

Opinnäytetyön tuloksena on toimiva skaalautuva ja turvallinen taustajärjestelmä. Tuloksena aikaansaatua toteutusta ei kuitenkaan oteta käyttöön, sillä suuri osa

taustajärjestelmän päätepisteistä tekevät kutsun Regatta-rajapintaan. FaaS-palvelussa suoritusaika maksaa ja funktiossa tehty kutsu toiseen rajapintaan lasketaan mukaan suoritusaikaan. Joten vaikka palvelun funktiota optimoitaisiin, on sen suoritusaika aina riippuvainen ulkopuolisen rajapinnan suorituskyvystä.

Minkälaiseen käyttöön Lambda sitten soveltuu erityisen hyvin? REST-rajapintojen osalta Lambda soveltuu lyhytaikaisiin operaatioihin, kuten suoraan kutsusta tehtyihin tietokanta-operaatiohin. Kun tietokantayhteys luodaan käsittelijän ulkopuolella, voidaan yhteys säilyttää kutsusta toiseen, kuten Lambda-esimerkissä alustettu RegattaDataService-apuluokka. Muita hyviä käyttötarkoituksia Lambdoille ovat esimerkiksi datan prosessointi, jonka tilanhallintaan voidaan käyttää Amazonin Step Functions -palvelua ja erityistä huomiota vaativista CloudWatch-hälytyksistä laukeavat ilmoitusviestit.

Toteutuksen suurimmat haasteet liittyivät Lambda-myöntäjään ja taustajärjestelmän CORS-tukeen. Kehitysprosessin aikana opin paljon Lambdan toiminnasta, API Gatewayn ominaisuuksista ja Serverless Frameworkin käytön eduista. Sain myös selvitettyä, millaisiin käyttötarkoituksiin kyseisiä teknologioita voitaisiin hyödyntää jatkossa.

LÄHTEET

Amazon Web Services 2017. Serverless Architectures with AWS Lambda [viitattu 12.5.2018]. Saatavissa: https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf

Amazon Web Services 2018a. Amazon API Gateway Pricing [viitattu 15.6.2018]. Saatavissa: https://aws.amazon.com/api-gateway/pricing/

Amazon Web Services 2018b. Amazon API Gateway: Developer Guide [viitattu 15.6.2018]. Saatavissa: https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-dg.pdf

Amazon Web Services 2018c. Amazon CloudWatch: User Guide [viitattu 20.6.18]. Saatavissa: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/acw-ug.pdf

Amazon Web Services 2018d. Amazon Web Services: General Reference [viitattu 16.6.18]. Saatavissa: https://docs.aws.amazon.com/general/latest/gr/aws-general.pdf

Amazon Web Services 2018e. AWS Lambda Pricing [viitattu 12.5.2018]. Saatavissa: https://aws.amazon.com/lambda/pricing/

Amazon Web Services 2018f. AWS Lambda: Developer Guide [viitattu 3.6.2018]. Saatavissa: https://docs.aws.amazon.com/en_us/lambda/latest/dg/lambda-dg.pdf

Amazon Web Services 2018g. The AWS Serverless Platform [viitattu 12.5.2018]. Saatavissa: https://aws.amazon.com/serverless/

Churchman, M. 2017. Containers vs. Serverless Computing [viitattu 31.8.2018]. Saatavissa: https://rancher.com/containers-vs-serverless-computing/

Express 2018. Express - Fast, unopinionated, minimalist web framework for Node.js [viitattu 3.8.2018]. Saatavissa: https://expressjs.com/

Google, Inc. 2018. Containers 101: What are containers? [viitattu 21.8.2018]. Saatavissa: https://cloud.google.com/containers/

Hefnawy, E. 2016. Serverless Code Patterns [viitattu 22.8.2018]. Saatavissa: https://serverless.com/blog/serverless-architecture-code-patterns/

Hefnawy, E. 2017. How to Test Serverless Applications [viitattu 9.7.2018]. Saatavissa: https://serverless.com/blog/how-test-serverless-applications/

HiQ 2018. Financial information [viitattu 22.8.2018]. Saatavissa: https://www.hiq.se/en/investor-relations/financial-information/

Malishev, N. 2018. Lambda Cold Starts, A Language Comparison [viitattu 31.8.2018]. Saatavissa: https://medium.com/@nathan.malishev/lambda-cold-starts-language-comparison-%EF%B8%8F-a4f4b5f16a62

Roberts, M. 2018. Serverless Architectures [viitattu 2.6.2018]. Saatavissa: https://martinfowler.com/articles/serverless.html

Roberts, M. & Chaplin, J. 2017. What is Serverless? Kalifornia: O'Reilly Media, Inc.

Sbarski, P. 2017. Serverless Architectures in AWS. New York: Manning Publications Co.

Serverless, Inc. 2018a. AWS - Credentials [viitattu 7.8.2018]. Saatavissa: https://serverless.com/framework/docs/providers/aws/guide/credentials/

Serverless, Inc. 2018b. AWS - Introduction [viitattu 28.6.2018]. Saatavissa: https://serverless.com/framework/docs/providers/aws/guide/intro/

Serverless, Inc. 2018c. Serverless AWS Lambda Events [viitattu 29.6.2018]. Saatavissa: https://serverless.com/framework/docs/providers/aws/events/

Serverless, Inc. 2018d. Testing [viitattu 29.6.18]. Saatavissa: https://serverless.com/framework/docs/providers/aws/guide/testing/

Serverless, Inc. 2018e. The Serverless Framework [viitattu 20.6.18]. Saatavissa: https://github.com/serverless/serverless

Serverless, Inc. 2018f. Why Serverless? [viitattu 20.6.18]. Saatavissa: https://serverless.com/learn/

Solanki, J. 2018. How Does a Serverless App Work? [viitattu 9.7.18]. Saatavissa: https://dzone.com/articles/how-does-a-serverless-app-work

Spoiala, C. 2015. Cloud offering: Comparison between IaaS, PaaS, SaaS, BaaS [viitattu 19.7.2018]. Saatavissa: https://assist-software.net/blog/cloud-offering-comparison-between-iaas-paas-saas-baas

Stephens, R. 2015. Beginning Software Engineering. New Jersey: John Wiley & Sons.

Stojanovic, S & Simovic A. 2018. Serverless Applications with Node.js. New York: Manning Publications Co.

Wittig M. & Wittig A. 2016. Amazon Web Services in Action. New York: Manning Publications Co.