Toomas Tero

Amazon AWS -pilvipalveluiden integrointi olemassa olevaan sovellukseen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

18.09.2015

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Toomas Tero Amazon AWS -pilvipalveluiden integrointi olemassa olevaan sovellukseen 32 sivua 18.9.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander Kehitysjohtaja Erkka Niemi

Pilvipalvelut ovat tulleet tärkeäksi osaksi melkein jokaista nykyistä web-sovellusta. Ne tarjoavat mahdollisuuden vähentää menoja, siirtää palvelimien ylläpidon pilvipalveluiden tarjoajalle, luoda satoja virtuaali-instansseja sekunneissa jne.

Pilvipalveluiden hankkijat tarjoavat nykyään laajan valikoiman palveluita ja niiden integraatioita toisiinsa. Ne tarjoavat mahdollisuuden kehittää sovelluksia riippumatta niiden suuruudesta – pienistä kotisovelluksista valtaviin sovelluksiin, jotka siirtävät miljoonia tunteja videoita. Tällainen on esimerkiksi Netflix.

Vuonna 2014 Amazon julkaisi Lambda-palvelun, joka mahdollisti palveluun pienen yksittäisen funktion viennin kokonaisen sovelluksen viennin sijaan. Lambda-palvelu tarjoaa hyvät mahdollisuudet skaalaukseen. Samalla palvelu eliminoi kuukausittaiset laskutukset: siinä laskutetaan vain ajokerrasta, RAM-muistin allokaatiosta ja funktion ajoajasta.

Sovelluksen kehitystä varten ja sen Amazon-palveluihin viennin helpottamiseksi on tehty Serverless-ohjelmistokehys.

Tämän insinöörityön tarkoitus on tutustuttaa lukija virtualisointiin ja Amazon AWS -pilvipalveluihin, kertoa KnoMe-sovellus ja siinä käytetyistä tekniikoista sekä analysoida, miten Lambdan ja muiden pilvipalveluiden integrointi on onnistunut Serverless-ohjelmistokehystä käyttäen.

Avainsanat	Amazon AWS, Serverless, Lambda, API, REST

Author(s) Title Number of Pages Date	Toomas Tero Integrating Amazon AWS Cloud Services into Existing Application 32 pages 18 September 2016
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Software Engineering
Instructor(s)	Simo Silander, Senior Lecturer Chief Development Officer Erkka Niemi

Cloud services have become the part of almost every web project in these days. They offer a good opportunity to reduce costs, shift hardware maintenance to service provider, deploy hundreds of virtual instances in seconds etc.

Cloud services providers now offer a large variety of services and integrations with each other. They offer a possibility to develop applications independently of application scale – from small home projects to tremendous projects like Netflix, which transfers millions hours of video.

In 2014 Amazon introduced a service called Lambda where instead of deploying the whole project as one you can deploy functions one by one. Lambda offers good scaling possibilities and is cheaper because instead of monthly-fee you are only charged when Lambda is triggered (although running time and RAM allocation is taken into account).

To simplify building projects around Lambda service, there has been developed Serverless framework. Serverless helps managing all Lambda functions depending on regions and stages.

The original goal of this thesis is to give a brief picture of what virtualization is, familiarize the reader to Amazon cloud services, introduce and describe KnoMe project and picture how integrating Lambda (and other) services using Serverless framework accomplished.

Keywords	Amazon AWS, Serverless, Lambda, API, REST

Sisällys

Lyhenteet

Lyhenteet

API	Application Programming Interface. Ohjelmoinnissa käytettävä rajapinta.
JSON	JavaScript Object Notation. Ominaisuus-avain-muodossa oleva tieto.
REST	Representational State Transfer. Arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen.
S3	Amazon Simple Storage Service. Pilvitallennusalusta.
EC2	Amazon Elastic Compute Cloud. Pilvipalveluinfrastruktuuri.

1 Johdanto

Pilvipalveluiden käyttö on lisääntynyt räjähtävästi viime vuosien aikana. Pilvipalvelut tarjoavat hyvin joustavan tavan hallita yrityksen resursseja ja tarvittaessa vähentää tai lisätä niitä nopeasti. Tällä hetkellä termi *pilvipalvelu* käsittää paljon muutakin, kuin mitä se käsitti vuosia sitten. Kaikkien palveluiden lisäksi koko ajan tulee uusia tuotteita, joista yritys voi valita itselleen sopivat.

Monilla yrityksillä on jo olemassa valmis tuote sisäiseen käyttöön tai ulkopuoliseen myyntiin. Vaikka yrityksen tuotteen siirtäminen pilvipalveluihin säästäisikin yritykseltä rahaa, infrastruktuurin kuluja ja ihmistyötunteja, yritys ei yleensä voi pysäyttää sovelluksen kehitystä pilvipalveluihin siirtymisen ajaksi. Usein yrityksen on kehitettävä tuotetta jatkuvasti, joten tuotteiden osia siirretään pilvipalveluihin tai aloitetaan käyttämään pilvipalveluita pikkuhiljaa integroimalla ne jo olemassa olevaan tuotteeseen.

Näin on tapahtunut myös KnoMessa. KnoMe on alun perin Siili Solutions Oyj -yrityksen sisäiseen käyttöön tarkoitettu CV-luettelomainen sovellus, jonka avulla yrityksen myyntiosasto pystyy paremmin allokoimaan konsultteja asiakkaille tai tiettyyn tehtävään. Osa Siili Solutions Oyj -yrityksen työntekijöistä aloitti KnoMen kehityksen vuonna 2011, ja se on pikkuhiljaa muuttunut vain CV-luettelomaisesta sovelluksesta enemmän toiminnallisuutta sisältäväksi sovellukseksi. Tullessani itse kehittämään KnoMea siihen päätettiin integroida Amazonin pilvipalveluita. Syinä oli Amazonin uuden Lambda-palvelun kokeilu, koodipohjan kehittäminen tulevaisuutta varten ja sovelluksen siirtäminen pikkuhiljaa pilveen.

Siili Solutions on suomalainen yritys, jossa työskentelee noin 430 ihmistä. Siili Solutions on ohjelmistotalo, jolla on toimipisteet Helsingissä, Oulussa, Berliinissä ja Wroclawissa. Siilin suurimpia asiakkaita ovat mm. Elisa, OP Pohjola, Nokia, Maa- ja metsätalousministeriö sekä Kesko Oyj. Siili tarjoaa asiakkaille konsultteja.

Tämän työn tarkoitus on tutkia, miten integrointi on onnistunut ja mihin ongelmiin olemme pääkehittäjän kanssa törmänneet integroinnin aikana. Insinöörityön toisessa luvussa kerron erilaisista pilvipalveluista ja niiden historiasta. Kolmannessa luvussa kerron Amazonin pilvipalveluista ja niiden hallintaa helpottavasta Serverless-

ohjelmistokehyksestä. Serverless helpottaa koodin ja konfiguroinnin vientiä Amazonin pilvipalveluihin, minkä tuloksena on kehittäjän työn helpottuminen.

Neljännessä luvussa kerron enemmän KnoMesta ja Amazonin pilvipalveluiden integroinnista. Kerron myös, mitä ongelmia integroinnin aikana on tullut vastaan. Viidennessä luvussa teen yhteenvedon integroinnista.

2 Pilvipalvelut

2.1 Virtualisointi

2.1.1 Yleistä

IT-maailmassa on aina ollut ongelmana resurssien jako. Yhdellä IT-yrityksellä on yleensä ollut erilaisia asiakkaita, jotka ovat tarvinneet eri määrän resursseja. Ennen virtualisointitekniikan syntyä yhden tietokoneen resurssia oli hankala jakaa monille asiakkaille, joten IT-talot joutuivat allokoimaan jokaiselle asiakkaalle tietyn määrän resursseja. Esimerkiksi yhtä web-palvelinta kohti saattoi olla yksi tietokone, johon saattoi samalla kuormituksesta riippuen sisältyä tietokanta-palvelin. Ongelmana tässä tapauksessa oli se, että web-sivun käyttäjämäärä voi kasvaa tai vähentyä monikertaisesti lyhyessä ajassa, joten nopeasta muutoksesta ja resurssien päivittämisestä jouduttiin kärsimään usein.

Virtualisointi on saanut IT-maailmassa aikaan ison muutoksen. Termi *virtualisointi* tarkoittaa yhden fyysisen resurssin pilkkomista moniksi loogisiksi resursseiksi [1]. Pilvipalveluiden tarjoajalle siitä on hyötyä, koska tarjoaja voi pilkkoa yhtä tehokasta fyysistä palvelinta useiksi pieniksi osiksi ja tarjota tietylle asiakkaalle vain sen verran resursseja kuin asiakas käyttää tai haluaa. Tarjoaja tarvitsee vähemmän fyysisiä palvelimia ja voi kontrolloida niitä paremmin. Tarjoaja voi myös tarjota pilvipalveluita entistä isommalle määrälle asiakkaita. Tällöin tarjoaja vähentää menojaan huomattavasti. Tilanteen muuttuessa allokoidut resurssit voidaan nopeasti päivittää loogiselle osalle. Virtualisointi myös parantaa tarjoajan kilpailukykyä, koska sen avulla tarjoaja voi nopeammin reagoida muuttuvaan kilpailutilanteeseen.

Virtualisointi parantaa myös pilvipalveluiden asiakkaan kilpailukykyä. Esimerkiksi käyttäjämäärän nelinkertaistuessa onnistuneen mainoksen ansiosta asiakas voi reagoida muutoksiin nopeasti ja päivittää oman palvelimensa resurssit sopiviksi. Lisäksi asiakkaan ei tarvitse enää niin paljon huolehtia fyysisistä palvelimista, koska se on osittain delegoitu pilvipalveluiden tarjoajalle.

Tarkoituksista ja valmistajasta riippuen virtualisointiohjelma voidaan luoda joko koneen käyttöjärjestelmäksi tai jo olemassa olevan käyttöjärjestelmän päälle (esim. Linux, Windows, OS X), joten virtualisointia voivat kokeilla kaikki siitä kiinnostuneet ihmiset. Ohjelmoinnissa käytetään yleensä jälkimmäistä tapaa eli asennetaan virtualisointiohjelma isäntäkoneen käyttöjärjestelmän päälle ja luodaan virtuaalikoneita johonkin fyysisellä koneella olevaan kansioon.

2.1.2 Virtualisointi kehityksen apuna

Ohjelmistot ovat nykyään valtavia ja käyttävät paljon olemassa olevia kirjastoja. Melkein jokainen ohjelmisto käyttää jotakin kirjastoa, joka yleensä itse käyttää toista kirjastoa. Näin ketju voi jatkua pitkäänkin. Jossakin tällaisen ketjun lenkissä voi tapahtua niin, että kirjasto on tehty vain tietylle käyttöjärjestelmälle. Tällöin esimerkiksi Windows-käyttöjärjestelmässä ei ole enää mahdollisuutta käyttää ohjelmistoa, joka käyttää Linux-käyttöjärjestelmälle tarkoitettua kirjastoa. Toinen ongelma voi esiintyä silloin, kun järjestelmää on käytetty kehitykseen kauan, ja on jo liian monimutkaista selvittää, mitä kirjastoja on käytetty ja mitä asennettuja riippuvuuksia ne käyttävät niiden sisällä.

Ohjelmointimaailmassa on tästä syystä syntynyt ilmaisu ”works on my machine” (toimii minun koneellani), joka tarkoittaa sitä, että tietyllä koneella toimiva sovellus ei edellä mainituista syistä toimi toisella koneella. Tätä ilmaisua käytetään muihinkin tilanteisiin liittyen.

Virtualisoinnista on tässä apua, koska sen avulla olemassa olevan käyttöjärjestelmän päälle on mahdollista asentaa ns. tabula rasa -käyttöjärjestelmä eli vasta asennettu järjestelmä. Sitä voidaan käyttää kehitykseen riippumatta siitä, millä käyttöjärjestelmällä fyysinen kone toimii. Tämä säästää kehittäjien aikaa ja yrityksen menoja, koska kehittäjät voivat keskittyä tuotteen kehitykseen, eikä heidän tarvitse enää taistella eri koneiden yhteensopivuuksien kanssa.

2.2 Pilvipalvelut

Termiä *pilvipalvelu* on nykyään vaikea määritellä yhdessä lauseessa, koska se ei enää tarkoita samaa kuin kymmenen vuotta sitten, ja raja katoaa pilvipalveluiden tarjoajien koko ajan lisätessä uusia palveluita. Tuntuu siltä, että *pilviksi* kutsutaan nykyään kaikkia sellaisia palveluita ja resursseja, jotka eivät ole käyttäjän fyysisessä ulottuvuudessa vaan jossakin päin internetiä, ”pilvessä”.

Alun perin termi tarkoitti palvelimen vuokrausta eli ei fyysisessä ulottuvuudessa olevaa palvelinta. Yritys saattoi vuokrata palvelinta esimerkiksi tunniksi, viikoksi tai vaikka vuodeksi. Käyttäjien määrästä riippuen yritys saattoi ostaa enemmän resursseja ja hiljaisena aikana luopua osasta resursseja. Tällöin yritys säästi rahaa ja pystyi käyttämään sitä muihin tarkoituksiin nostamalla kilpailukykyasemaansa. Nykyään pelkkien palvelimien lisäksi on tehty iso kokonaisuus erilaisia palveluja, joita yritys voi käyttää yksitellen tai integroimalla niitä toisiinsa. Pilvipalveluiden tarjoaja yleensä helpottaa integrointimenettelyä tarjoamalla apukirjastoja. Tästä on esimerkkinä Amazon AWS, jonka pilvipalveluista kerron enemmän luvussa 3.

2.2.1 Konfiguraatiotyökalu

Vaikka pilvessä olevia instansseja on helppoa päivittää, luoda ja tuhota, ohjelmistojen asennus voi kestää pitkään ja olla hankalaa. Esimerkiksi tätä varten voidaan tehdä apuskriptejä, joiden avulla on helppoa asentaa ohjelmia tai kirjastoja Linux-käyttöjärjestelmässä. Yksinkertainen skripti, joka asentaa Firefox-verkkoselaimen, voi näyttää tältä:

```
sudo apt-get install firefox
```

Koodiesimerkki 1. Firefox-selaimen asennus pakettihallinnan avulla Linux-ympäristössä.

Vaikka skripti koodiesimerkissä 1 näyttää helpolta ja yksinkertaiselta, tällä tavalla ei ole kovin paljon mahdollisuuksia vaikuttaa siihen, mitä, miten ja milloin skripti tekee (apt-get-pakettihallinnan tarkkaa toiminnallisuutta ja asetuksia lukuun ottamatta). Nykyaikaisessa ohjelmistossa on paljon muuttujia, joita pitää muuttaa riippuen siitä, onko ohjelmisto kehitys- vai tuotantoympäristö, ja onko se kehitystä varten paikallisella koneella vai pilvessä. Erilaiset ympäristöt tarvitsevat myös erilaisia muuttujia, kuten esimerkiksi Amazon AWS- ja Microsoft Azure -tunnukset, Google Analytics -koodi,

uniikkiotsikko web-pyynnössä jne. Kehittäjän kannattaa esimerkiksi asentaa kirjastojen dokumentaatiot ja testitkin paikallisesti niiden koosta riippumatta. Tuotannossa halutaan yleensä mahdollisimman pieni kirjasto ilman lisätiedostoja, kuten kirjaston dokumentaatiota tai testejä. Toinen esimerkki on JavaScript-koodin minimointi. Kehitysympäristössä halutaan analysoida oikeaa koodia, kun taas tuotannossa on laitettava minimoitu versio.

Tätä ongelmaa varten on kehitetty omat työkalut, kuten Ansible tai Chef. Jälkimmäistä käytämme KnoMe-sovelluksessa paikallisesti virtuaalikoneessa (ja Amazonin instanssissa) tarvittavien kirjastojen ja koodien asennukseen, tiedostojen ja kansioiden luomiseen ja oikeuksien muuttamiseen sekä palvelimen starttaukseen. Sen lisäksi Chef-lisämoduulien avulla pystytään luomaan konfiguroituja instansseja Amazonissa ennen asennuksen alkua. KnoMessa Chef on konfiguroitu niin, että koko sovelluksen pystytys tapahtuu yhdellä komentorivillä.

2.2.2 Pilvipalveluiden tarjoajat

Amazon

Amazon on ehkä kaikista tunnetuin pilvipalveluiden tarjoaja. Lisäksi Amazonin nimen mainetta on vahvistanut se, että Amazon on myös suurin ja tunnetuin internetkauppa. Toisaalta yritys tunnistetaan juuri kaupasta, eikä suurin osa IT-maailman ulkopuolella olevista ihmisistä tiedä, että Amazon tarjoaa myös ison valikoiman pilvipalveluja. Koko Amazon-internetkaupan infrastruktuuri toimii nyt käyttäen Amazonin tarjoamia pilvipalveluita.

Kuten kuvasta 1 näkyy, Amazon tarjoaa tarvittavan määrän palveluja ja niiden integrointia. Nämä palvelut ja integrointi riittävät monille yrityksille palveluiden ja tuotteiden luomiseen osittain tai suoraan pilvissä. Pilvipalveluiden asiakas voi käyttää yhtä tai useampaa tuotetta palvelimien luontiin, tekstinä tietokantaan tai tiedostoina S3-palveluun tallennukseen, tapahtumalaukaisuun, reititykseen, esineiden internetiä varten, mobiiliohjelmointiin jne.

Tämän lisäksi jokaisella tuotteella on omat ominaisuutensa. Esimerkiksi tietokannoilla ja S3-levyjärjestelmällä on mahdollisuus replikoida itseään eli kopioitua automaattisesti

yhteen tai moniin alueisiin. Se nostaa palvelun hintaa, mutta takaa samalla datan säilytyksen riippumatta datakeskuksessa tai sen lähellä tapahtuvista onnettomuuksista.

Amazon tarjoaa pilvipalveluiden lisäksi hyvän mahdollisuuden liittää pilvipalveluita toisiinsa. Se on todella tärkeää, koska sovellus käyttää yleensä useaa palvelua samaan aikaan. Web-sovellukset, jotka eivät käytä API-rajapintoja ja tietokantaa, ovat hyvin harvassa. Tätä varten Amazon on kehittänyt Amazon SDK -ohjelmistokehityspaketin (software development kit), joka tarjoaa valtavan valikoiman rajapintoja kaikkia palveluita ja niiden integrointia varten. Tavallisten poisto-, luku-, lisäys- ja muokkausoperaatioiden lisäksi Amazon tarjoaa mm. mahdollisuuden (luettelen vain KnoMessa käytetyt) poistaa S3-levyjärjestelmiä, asettaa niille laukaisukonfiguraatioita, määrittää replikaatiot, luoda ja poistaa uusia taulukkoja tietokannassa, asettaa taulukolle kirjoitus- ja lukukapasiteetti jne. Amazon SDK:ssa on jokaista palvelua varten oma nimiavaruutensa yleisillä ohjelmointikielillä, kuten JavaScript, Java, .Net ja Python [2]. Koska tuettujen ohjelmointikielten listasta löytyy suurin osa maailman eniten käytetyistä ohjelmointikielistä [3], melkein jokainen sovellus, johon Amazon-pilvipalvelut haluttaisiin integroida, hyödyntäisi SDK:tä kehitystyön helpottamiseen.

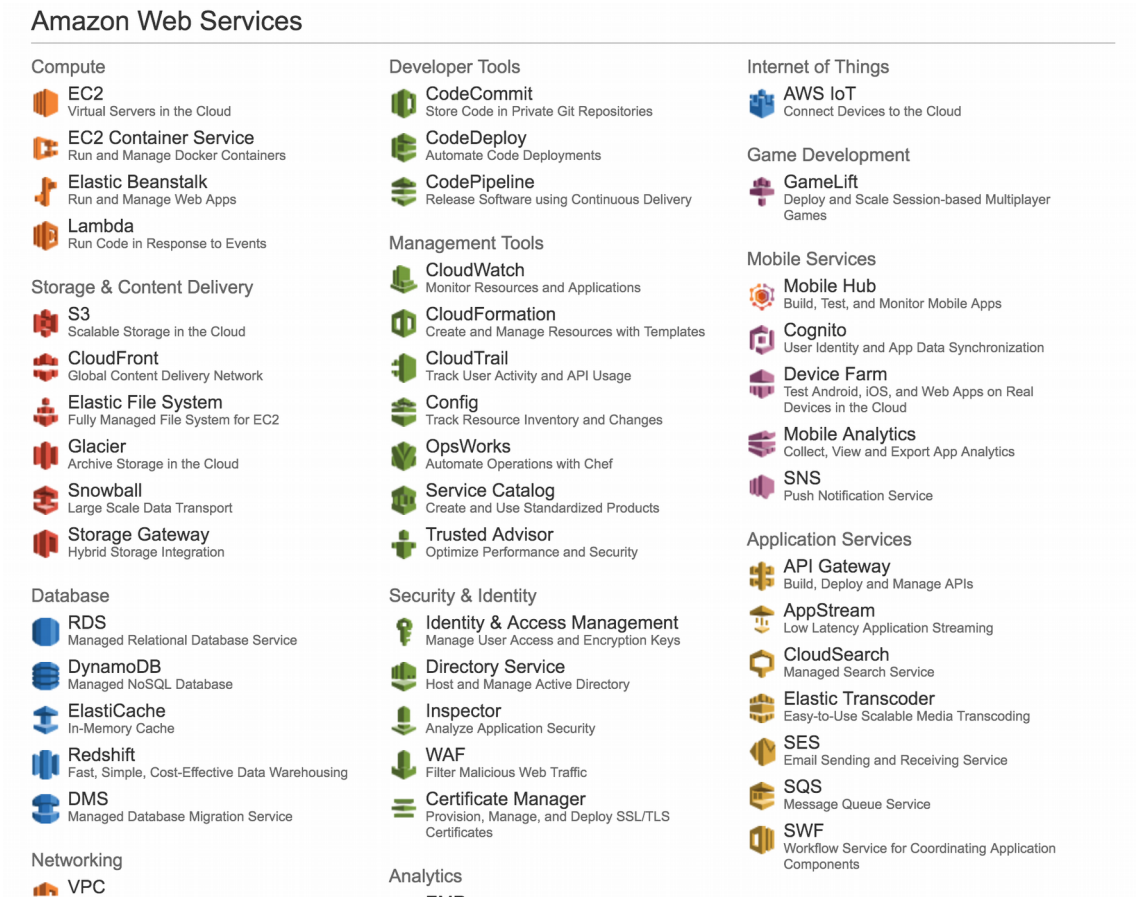
Kehitystyön helpottamisen lisäksi ohjelmistokehityspaketti on omaan implementaatioon verrattuna hyvä myös siinä mielessä, että se on testattu hyvin paketin tarjoajalla, ja kaikki tarjoajalla tai muilla esiintyneet virheet ja sekavat kohdat on otettu huomioon. Ja vaikka julkaistua versiota ei olekaan mahdollista enää muuttaa, Amazon SDK sisältää kaikki versiot, ja SDK:hon on tehty mekanismi, jolla voi asettaa haluamansa version käyttöön. Esimerkiksi NodeJS SDK:ssa tämä on ratkaistu todella kätevällä tavalla. Initialisoinnin aikana käyttäjä voi asettaa version parametrina esimerkiksi seuraavalla tavalla:

```
const dynamodb = new AWS.DynamoDB({apiVersion: '2012-08-10'});
```

Koodiesimerkki 2. JavaScript-ohjelmointikielessä SDK:n objektin initialisointi ES6-syntaksia käyttäen.

Koodiesimerkissä 2 käytetty tapa on viisas ja estää kätevästi sisällä olevien funktioiden keskinäisten konfliktien syntymistä. Valitettavasti ei ole mahdollista tietää etukäteen, mikä arkkitehtuurinen rakenne on paras johonkin API-rajapintojen suunnitteluun. Tämän takia jokainen ohjelmistokehityspaketin tarjoaja yrittää analysoida, miten

pakettia käytetään, ja tutkia palautteiden perusteella heikoimmat ja eniten aikaa vievät kohdat. Jos tarjoaja esimerkiksi näkee, että tiedon luku tietokannasta on hidasta, se voi käyttää ihmisresursseja sen parantamiseen. Tuloksena voi olla lukualgoritmin parannus, jolloin luku alkaa viedä vähemmän aikaa. Näistä syistä on todella tärkeää, että käyttäjällä on mahdollisuus vaihtaa versio haluamakseen, ja että hän tiedostaa tämän mahdollisuuden. Täytyy huomata, että Amazonin tarjoama SDK (ainakin JavaScript-ohjelmointikieltä käyttävä) eroaa tavallisesta semanttisesta versioinnista, jossa versionumerot esitetään pääversionumero, alaversionumero ja korjausversionumero pisteellä erotettuina eli esimerkiksi 3.2.1. Amazon käyttää version numerona päivämäärää, jona SDK on julkaistu.



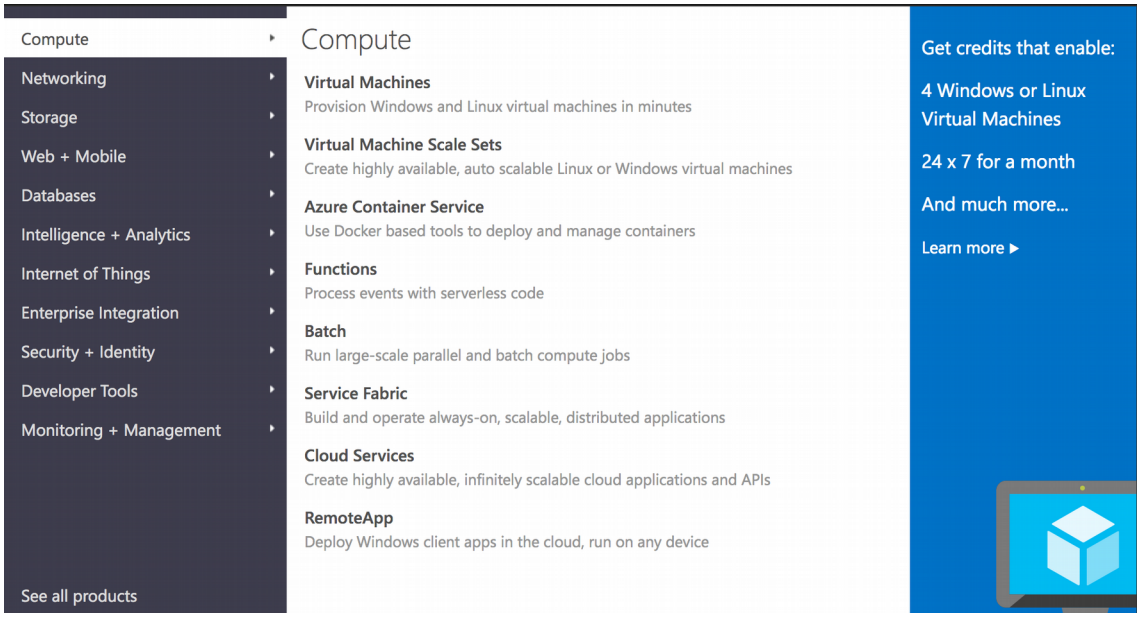
Kuva 1. Osa Amazon AWS:n tarjoamista pilvipalveluista. KnoMe-sovelluksessa tärkeimmät ovat EC2, Lambda, DynamoDB, Api Gateway, S3 ja CloudWatch.

Microsoft

Microsoftin Azure-pilvipalveluiden sivulle [4] tultaessa voidaan nähdä paljon suurin piirtein tuttuja kohtia. Microsoft Azure on Amazon AWS -pilvipalveluiden suora kilpailija, joten se tarjoaa melkein samanlaiset pilvipalvelut kuin Amazon. Siilillä käytetään sekä Amazon AWS- että Microsoft Azure -pilvipalveluita. Amazon AWS -palveluita käytetään kuitenkin enemmän.

Microsoft Azure voi olla miellyttävämpi kehittäjille, jotka haluavat käyttää C#-ohjelmointikieltä ja sille kehitettyjä kirjastoja, integroida omat ratkaisunsa SharePoint-järjestelmään tai vaikka kehittää ohjelmia tai pelejä XBOX-konsolia varten. Koska XBOX on Microsoftin omistama, se tarjoaa valmiit työkalut kehitystä varten.

Microsoft ja Amazon käyvät keskenään kovaa kilpailua. Molemmat tarjoavat tilapäisen ilmaisen kokeilun, jonka avulla yritys tai henkilö voi päättää, kumpi vaihtoehdoista sopii paremmin.



Kuva 2. Microsoft Azure -pilvipalveluita. Microsoft on Amazonin kilpailija, joten niiden palvelut ovat suurin piirtein samanlaisia joitakin tiettyjä ominaisuuksia lukuun ottamatta.

3 Amazon AWS -pilvipalvelut ja Serverless-ohjelmistokehys

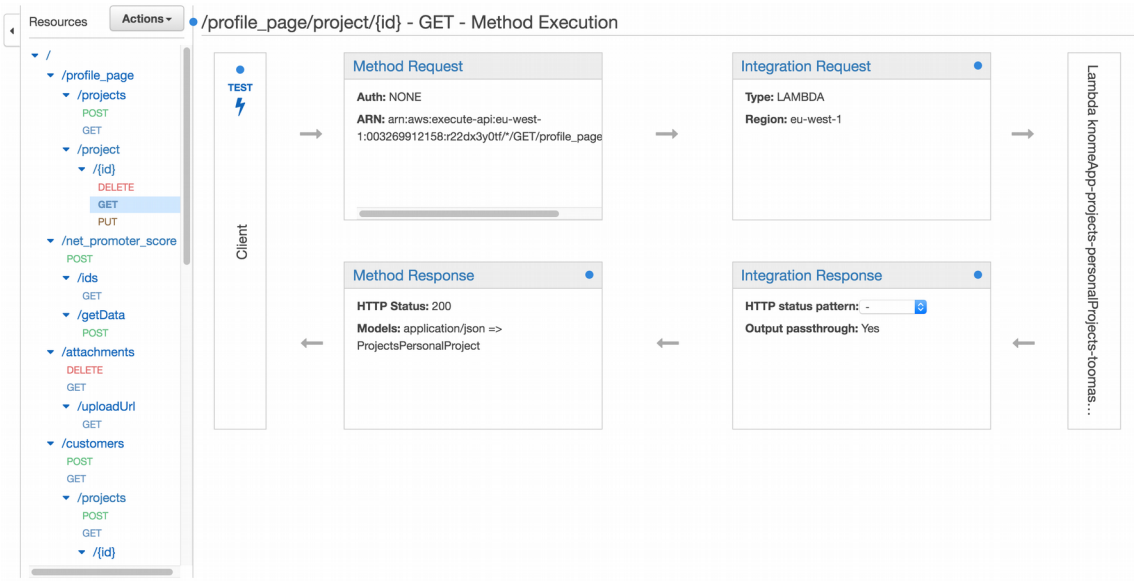
3.1 Amazonin AWS -pilvipalvelut

Yleistä

Amazonilla on niin laaja palvelininfrastruktuuri, että se pystyy tarjoamaan pilvipalvelut kuormituksesta riippumatta. Netflix [5] on hyvä esimerkki isosta yrityksestä, joka toimittaa noin 100 miljoonaa tuntia videoita päivässä. Amazonin avulla se pystyy hetkessä luomaan ja tuhoamaan satoja virtuaalikoneita vaatimusten tyydyttämiseksi.

API on tärkeä osa nykypäiväistä ohjelmistoa, joten Amazon tarjoaa API Gateway -palvelun, jonka avulla jokainen kehittäjä voi helposti päästä tekemään API-rajapintoja, analysoida niitä informatiivisessa näkymässä, rajoittaa tulevan datan muotoa ja tarvittaessa muuttaa rajapintoja sopivaan muotoon. Palvelun web-sivulla näkyy kätevästi koko API-rakenne, ja tarvittaessa näkyviin voi valita yhden API-rajapinnan päätepisteen, josta löytyvät sen kaikki vaiheet ja tiedot. Jos kehittäjä haluaa muuttaa tätä päätepistettä, hänellä on mahdollisuus päästä muuttamaan ja sen jälkeen testaamaan päätepistettä vaiheittain.

Kuvassa 3 näkyy oikealla puolella neljä laatikkoa. Ne ovat vaiheita, joiden kautta web-pyyntö kulkee Api Gateway -palvelussa. Integration Request -laatikosta tiedämme, että tämä pyyntö käynnistää ja kutsuu Lambda-palvelun, joka käsittelee pyynnön ja palauttaa vastauksen implementaatiologiikasta riippuen. Lambda ei ole ainoa palvelu, jonka voi linkittää päätepisteeseen. KnoMessa käytetään ehkä hieman enemmän Amazonin EC2-virtuaalipalvelimia eli välitetään API-pyynnöt uniikkiavaimen perusteella johonkin virtuaalipalvelimeen, joka käsittelee ne ja palauttaa vastauksen. Vaikka EC2- ja Lambda-palvelut voivat kuulostaa samankaltaisilta, niissä on eroja, joista kerron myöhemmin. API Gateway -palvelulle ei ole väliä, mikä palvelu pyörii ja vastaanottaa pyyntöjä, koska se toimii vain ”porttimiehenä”, joka tarkistaa kutsun ja välittää sen eteenpäin. Kun vastaus välitetyltä palvelulta palaa, API Gateway tarkistaa sen omien sääntöjensä mukaan ja muuttaa sen sopivaan muotoon. Säännöt löytyvät Integration Response- ja Method Response -laatikoista. Integration Response -laatikko katsoo, mikä sisäisen vastauksen statuskoodi on ja muuttaa Method Response -laatikossa olevan mallin sopivaksi. Jos on esimerkiksi tapahtunut sisäinen virhe, Api Gateway voi muokata vastausta lisäämällä siihen mukautettuja viestejä.



Kuva 3. API Gateway -palvelun web-sivu. Vasemmalla näkyy API-rakenne ja oikealla yhden rajapinnan päätepisteen kaikki vaiheet.

EC2-palvelu tarjoaa virtuaalikoneiden instansseja. EC2-palvelua käytetään vanhaa palvelinta varten, ja sillä pyörii Node.js -palvelin.

Amazon Lambda on suhteellisen uusi, vuonna 2014 julkaistu palvelu, jonka avulla koodia ajetaan jonkin tapahtuman pyynnöstä. Lambda on saanut nimensä funktionaalisen ohjelmoinnin maailmasta. Siinä lambdaksi kutsutaan anonyymia ja pientä, usein jopa yhdellä rivillä määritettyä funktiota, joka saa syötteen, operoi sitä ja palauttaa arvon. Amazon Lambda on tärkein palvelu, joka haluttiin integroida KnoMe-sovellukseen. Lambda on siinä mielessä mielenkiintoinen, että koodia ajetaan ilman tiettyä palvelinta, johon viitataan IP-osoitteella tai tunnuksilla. Viittaaminen tapahtuu jollakin toisella tavalla, kuten Api Gateway -palvelun avulla. Satunnaisesti valittu palvelin ajaa koodia, ja ajaminen loppuu vastaus-funktion kutsumisen jälkeen, minkä jälkeen palvelinta voi käyttää jokin muu Lambda-funktio. Lambdan ja EC2-palvelun laskutuksen erona on se, että koska Lamba on pieni koodipätkä, joka ajetaan tapahtuman laukaistessa, siinä laskutetaan vain funktion ajokestosta, RAM-muistin allokaatiosta ja ajokerrasta. EC2-virtuaalikoneen instanssista taas laskutetaan päällä olevien tuntien mukaan riippumatta siitä, onko palvelimessa oleva ohjelma tekemässä jotakin hyödyllistä vai onko koko instanssi täysin tyhjä ilman mitään resurssien kuormitusta.

Lambdassa on yksi tulokohta (entry point), joka käynnistää Lambda-funktion, ja loppufunktio, johon koodin ajo (ainakin kehittäjän kontrolloima) loppuu. Tämä asettaa ohjelman arkkitehtuurille tiettyjä rajoituksia. Koska Lambdan pitää olla nopea ja loppua heti, kun tarvittu toiminnallisuus on suoritettu, tuntuu luontevalta tehdä pieniä funktioita. Tähän liittyvät vahvasti mikro- ja nanopalveluarkkitehtuurit. Koska nanopalveluarkkitehtuuri on vielä pienempiin osiin jaettu mikropalvelut-arkkitehtuuri, tarkastelen tässä jälkimmäistä. Mikropalveluarkkitehtuurilla tarkoitetaan sovellusta, joka rakentuu pienistä, riippumattomista osista eli palveluista. Sovelluksessa on yksi pääpalvelu, joka käyttää kaikkia pieniä palveluita, mutta samalla jokainen palvelu voi toimia muiden palveluiden kanssa. Tästä huolimatta jokainen palvelu on riippumaton muista palveluista ja pääpalvelusta. Yleensä palvelut keskustelevat keskenään REST-rajapinnalla.

Tämä jakaaminen voi tuntua erikoiselta, eikä välttämättä ole heti selvää, miksi arkkitehtuuria haluttaisiin käyttää. Koska arkkitehtuurissa palvelu on riippumaton, sitä voidaan helposti päivittää, muuttaa tai kirjoittaa uudelleen jollakin toisella ohjelmointikielellä, jos siihen on pätevä syy. Palvelun ei tarvitse olla tehty jollakin tietyllä tavalla, kunhan se tarjoaa tarvittavat rajapinnat. Toisin sanoen palvelua voidaan esimerkiksi muuttaa niin, että se koostuu kahdesta palvelusta. Palvelu voidaan siirtää eri paikkaan, kuten Amazon AWS -pilvestä Microsoft Azure -pilveen, eikä tämä vaikuta pääpalveluun mitenkään. Arkkitehtuuriin liittyy myös skaalautuvuus. Yksi palvelu on mahdollista satakertaistaa ja tasapainottaa kutsuja kuormituksesta riippuen (mihin voidaan tarvita kuormituksen tasapainottaja).

Lambda on joustava, ja siinä voidaan käyttää eri arkkitehtuureja, mutta mikropalveluarkkitehtuuri vahvasti liittyy Lambdaan. Tavallisin tapa Api Gateway -palvelua käytettäessä on, kun yhteen REST-päätepisteeseen liittyy yksi Lambda-funktio, joka suorittaa bisneslogiikan ja loppuu. Tämä Lambda-funktio eli palvelu voidaan muuttaa eri ohjelmointikieleen (esim. JavaScriptistä Javaan), tai voidaan tehdä kaksi erillistä Lambda-funktiota, joita kolmas Lambda-palvelu käyttää. Amazon huolehtii skaalautuvuudesta, eli Amazon voi käynnistää satoja Lambda-funktioita, jos käyttäjän määrä kasvaa.

Perinteistä sovellusta on vaikea muuttaa mikropalveluarkkitehtuuriksi. Työmäärä riippuu sovelluksen koosta ja arkkitehtuurista, ja pahimmassa tilanteessa koko sovellus ja sen arkkitehtuuri pitää miettiä ja kehittää uudestaan. Tämä pätee hyvin usein vanhoihin ja isoihin projekteihin. Pilvipalveluiden tarjoajat lisäävät koko ajan uusia

palveluita ja helpottavat olemassa olevien käyttöä, joten usein sovellus hyötyisi mikropalveluarkkitehtuurista melko paljon. Samalla muutos helpottaa kehitystä ja laajennusta tulevaisuudessa.

Amazon Dynamo toimii tietokantana, johon esimerkiksi Lambda voi tallentaa dataa. Dynamo on niin sanottu NoSQL-tietokanta. NoSQL-tietokannoksi kutsutaan jokaista relaatiomallista poikkeavaa tietokantaa. Hyviä ja tunnettuja esimerkkejä ovat Amazonin DynamoDB, MongoDB ja CouchDB. Yleensä NoSQL-tietokanta tallentaa tiedot JSON-formaatissa ja riippuen tietokannan toteutuksesta tarjoaa erilaisia toiminnollisuuksia työskennellä ja käsitellä tallennettua dataa. Esimerkiksi vaikka MongoDB on NoSQL-tietokanta, se tarjoaa mahdollisuuden tehdä relaatioita ja käyttää niitä hakuoperaatioissa. Amazonin Dynamo ei tarjoaa relaatioita, joten tietojen yhdistäminen on tehtävä käyttäjän tasolla. NoSQL-tietokanta on parempi vaihtoehto silloin, kun kehityksen aikana ei ole vielä etukäteen selvää ymmärrystä, missä muodossa ja minkä tyyppistä data tulee olemaan. Relaatiotietokannoissa pitää etukäteen mallintaa, millaista dataa tietokannan taulukkoon tallennetaan. Esimerkiksi tuttu SQL-tietokannan taulukon määrittely voisi näyttää tältä:

```
CREATE TABLE Persons
(
    Id int,
    FirstName varchar(255),
    LastName varchar(255),
    Phone int
);
```

Koodiesimerkki 3. Uuden taulukon määrittely SQL-tietokannassa.

Olemassa oleva ja taulukkoon tuleva data on tiukasti sidottu määrittelyyn. Koodiesimerkissä 3 id-kenttä määritettiin numeroksi, etunimi-kenttä merkkijonoksi, sukunimi-kenttä merkkijonoksi ja puhelinnumero-kenttä numeroksi. Jos tällaiseen taulukkoon yritettäisiin lisätä osoite-tietue, tietokanta heittäisi virheen siitä, että osoite-tietue ei ole olemassa. Samalla puhelinnumeroksi ei sovi objekti, boolean-arvo tai merkkijono (jos tietokannan tasolla ei tapahdu automaattista konvertointia).

NoSQL-tietokanta on yleensä skeematon, eli siihen ei tarvitse etukäteen tehdä mitään määritystä. Amazon Dynamossa ainoa pakollinen kohta on uniikki id-kenttä, joka voi olla yksi tai useampi tietue. Datatyyppinä Dynamo hyväksyy numerot, merkkijonot, boolean-arvot, taulukot, JSON-objektit ja binääridataa. Binääridatassa on rajoituksena koko, joka ei voi olla enemmän kuin 400 kilotavua. Vaikka Dynamo tarjoaa mahdollisuuden tiedostojen tallentamiseen, se ei ole sen ideana. Erilaisia ja erikokoisia tiedostoja varten Amazon tarjoaa S3-palvelun, johon voi tallentaa tiedostoja ja tehdä niille muitakin operaatioita. S3-palvelussa olevien tiedostojen yhdistäminen Dynamossa olevaan objektiin vaatii enemmän lisätyötä, sillä S3 tarjoaa monta tapaa hakea ja tallentaa tiedostoja.

Dynamo ei tarvitse skeemaa, joten jos asettaisimme id-kentäksi merkkijonon ja sitten tallentaisimme Persons-taulukoon dataa, se voisi näyttää esimerkiksi seuraavalta:

```
{
    ”FirstName”: ”Maija”,
    ”LastName”: ”Meikäläinen”
}
```

Koodiesimerkki 4. Esimerkki json-formaatissa olevasta datasta.

Vaikka koodiesimerkissä 4 ei ole kovin paljon mitään tietoa, tällä tavalla ohjelma voi esimerkiksi alustaa valmiiksi etu- ja sukunimen uniikki-id:ksi (ottamatta huomioon tämän idean järkevyyttä), jotta kukaan ei myöhemmin voisi varata kyseisen henkilön tietoja. Kun uusia toiminnollisuuksia kehitetään ja uusia data-kenttiä lisätään ohjelman kasvaessa, Maija Meikäläinen -objekti tietokannassa voi muuttua seuraavaksi:

```
{
    ”FirstName”: ”Maija”,
    ”LastName”: ”Meikäläinen”,
    ”Age”: 18,
    ”Married”: false,
    ”Attachments”: [
        {
            ”S3Url”: ”…”,
            Tags: [”…”, ”…”],
            …
        },
```

```
        {…}
    ],
    "Events": {
        "Updated": {
            "Id": "…",
            "Timestamp": 1469867696142,
            …
        },
        "Created": {…}
    }
}
```

Koodiesimerkki 5. Kehityksessä laajennettu versio koodiesimerkistä 4.

Kuten näemme, NoSQL on joustava ja vaatii vähemmän työtä datan päivittämisessä. Myöhemmin, kun ohjelman arkkitehtuuri on selvä ja datan muoto on vakiintunut, voidaan päivityksen jälkeen siirtyä käyttämään SQL-tietokantaa.

S3 on ehkä viimeisin palvelu, joka KnoMessa otettiin käyttöön. Amazon S3 on pilvitallennusalusta, johon on mahdollista tallentaa erilaisia ja erikokoisia tiedostoja. S3-palvelua voidaan ajatella sellaisina kokoelmina verkkolevyjä (bucket), joita voidaan luoda ja tuhota. KnoMessa S3-palvelua käytetään liitteiden ja asiakkaiden logojen tallentamista ja varmuuskopiointia varten. S3 tarjoaa monta mahdollisuutta ladata tiedostoja itseensä. Eräs tapa on käyttää SDK:n tarjoamaa upload-funktiota, mutta KnoMessa se tarkoittaisi sitä, että tiedosto ladataan ensin KnoMen palvelimeen ja vasta sen jälkeen S3-palveluun. Tätä varten Amazon on kehittänyt ns. allekirjoitetun linkin (signed url). Se on pitkä ja uniikki URL-linkki, jolla on tietty eloaika sekä toiminto, johon linkin käyttäjällä on oikeus. Linkki generoidaan SDK:n avulla ja parametriksi on aina pakko laittaa toiminto, joka linkin avulla tehdään. KnoMen tapauksessa käyttäjällä on vain kaksi toimintoa, joille generoidaan linkit: tiedoston lataaminen ja tiedoston poisto. Esimerkiksi kun käyttäjä yrittää ladata uuden liitteen asiakasprojektille, koodi kutsuu Amazon SDK:n funktion ”getSignedUrl”, johon tulee ensimmäisenä parametrina lataustoiminto eli tässä tapauksessa ”putObject”. Esimerkiksi:

```
…
return new Promise((resolve, reject) => {
    S3.getSignedUrl('putObject', params, (err, url) => {
        if (err) {
```

```
            return reject(handleError(err));
        }
        resolve(url);
    });
})
…
```

Koodiesimerkki 6. Amazon SDK:n getSignedUrl-funktion kutsu.

Jos linkin generointi on suoritettu onnistuneesti, takaisinkutsufunktio (callback function) palauttaa uniikkilinkin. Muussa tapauksessa se palauttaa virhe-objektin. Tallentamista varten uniikkilinkin generoinnissa on etuna esimerkiksi se, että käyttäjä voi ladata tiedoston koosta riippumatta omaa selainta käyttäen, joten tämä toiminnallisuus ei turhaan kuormita palvelinta. SDK:n funktion kutsumisessa ja linkkien generoinnissa pitää olla varovainen, koska jos käyttäjälle annetaan väärän toiminnon linkki, se voi aiheuttaa paljon haittaa. Uniikki linkki on esimerkiksi mahdollista generoida toiminnoille kuten deleteBucket tai deleteObjects, jotka poistavat koko tallennuslevyn (bucketin) tai vastaavasti kaikki sisällä olevat tiedostot.

Edellä esitetyn lisäksi Amazon tarjoaa mahdollisuuden asettaa jokaista toimintoa varten tapahtumakäsittelijä, joka voi olla esimerkiksi Lambda-funktio, ja joka laukaistaan, kun toiminto tietylle palvelulle on suoritettu. S3-palvelussa tapahtumakäsittelijän voi asettaa vaikka kun tiedosto on ladattu tai poistettu S3-tallennuslevystä. Tämä on hyödyllinen toiminnollisuus, koska allekirjoitetussa linkissä on ongelmana se, että palvelin ei voi tietää, milloin lataus on onnistunut, eikä täten voi päivittää entiteettiä tietokannassa. KnoMessa tapahtumakäsittelijän avulla kutsuttu Lambda-funktio osaa tiedoston latauksen tai poiston jälkeen päivittää tietokannassa olevan entiteetin.

Palveluja käytetään todella paljon ja muutetaan ja laajennetaan koko ajan, eikä ole enää mahdollista muistaa, missä, miten ja mitä palveluja pitää päivittää. Amazon ratkaisee tämän ongelman CloudFormation-palvelulla. Palvelu tarjoaa mahdollisuuden yhdistää Amazonin muita palveluja yhteen pinoon (stack), josta tehdään mallitiedosto (template). Tätä mallia käytetään myöhemmin sinikopiona eli jokainen voi hetkessä luoda, päivittää ja poistaa oman pinonsa mallia käyttämällä. Etuna on, että jokainen käyttäjä saa aina tasan sellaisen infrastruktuurin, joka on määritetty mallissa. Mallia voi vielä käyttää kohdassa 2.2.1 mainitun konfiguraatiotyökalun kanssa. Sillä tapahtuu sekä automaattinen pilvipalveluiden infrastruktuurin pystytys että yksittäisen instanssin automaattinen ohjelmien ja kirjastojen lataaminen ja kääntäminen.

CloudFormation-malli voi olla tavallinen txt- tai json-tiedosto. Sen lisäksi on mahdollista käyttää myös muitakin formaatteja, jotka lopuksi käännetään json-formaatiksi. Yksi sellaisista on yaml-formaatti, joka on vähän lyhyempi ja selkeämpi verrattuna json-formaattiin. KnoMessa käytetään osittain json- ja osittain yaml-formaattia.

Koodiesimerkki 7 on osa CloudFormation-konfiguraatiotiedostosta. Vaikka alussa saattaa tuntua, että konfiguraatio on liian pitkä ja monimutkainen (esimerkiksi KnoMen CloudFormation-konfiguraatiotiedoston pituus on yli 500 riviä), ei voi olla huomaamatta, kuinka paljon se antaa verrattuna infrastruktuurin pystytykseen ja kaikkien oikeuksien tai asetuksien muuttamiseen käsin.

```
…
"CustomersTable": {
    "Type": "AWS::DynamoDB::Table",
    "Condition": "CreateInDevEnv",
    "DeletionPolicy": "Delete",
    "Properties": {
        "AttributeDefinitions": [
            {
                "AttributeName": "id",
                "AttributeType": "S"
            }
        ],
        "ProvisionedThroughput": {
            "ReadCapacityUnits": 1,
            "WriteCapacityUnits": 1
        },
        "TableName": "$${tableCustomers}"
    }
},
"s3AttachmentsBucket": {
    "Type": "AWS::S3::Bucket",
    "Condition": "CreateInDevEnv",
    "DeletionPolicy": "Delete",
    "Properties": {
        "BucketName": "$${s3AttachmentsBucketName}",
        "AccessControl": "Private"
    }
```

```
},
…
```

Koodiesimerkki 7. Osa mallitiedostosta (template file), jolla luodaan tai poistetaan Amazon AWS:n pino (stack).

Esimerkkikoodi 7 sisältää monta mielenkiintoista kohtaa. Ensinnäkin CloudFormation-palvelu osaa itse lukea ja sijoittaa muuttujia, joten tiedostossa saa olla ehtolauseita. Esimerkkikoodissa 7 Condition-kohdassa on ehtolauseena, että tietty palvelu (tässä tapauksessa DynamoDB:n taulukko ja S3-tallennuslevy) poistetaan ja luodaan uudestaan vain, jos ollaan kehitysympäristössä. Toinen mielenkiintoinen kohta on se, kuinka laajat mahdollisuudet CloudFormation tarjoaa. Esimerkkikoodi 7 on vain pieni osa siitä, mitä asetuksia, rajoituksia, tapahtumalaukaisu-sääntöjä ja muuta voi asettaa [6].

Kolmas kohta on omat muuttujat. Cloudformation sijoittaa muuttujan paikalle jossakin muualla aiemmin annetun arvon. Esimerkkikoodissa 7 DynamoDB:n taulukon nimeksi tulee

```
$${tableCustomers}
```

Koodiesimerkki 8. Muuttuja CloudFormation-mallitiedostossa (template file).

KnoMessa eräässä sovellukseen liittyvässä json-tiedostossa on laitettu:

```
"tableCustomers": "${project}-customers-${stage}"
```

Koodiesimerkki 9. Asiakas-taulukon nimeksi tulee tekstin ja toisten muuttujien yhdistelmä. Amazon sijoittaa muuttujien paikoille sovelluksen nimestä ja vaiheesta (stage) tulevat arvot.

3.2 Serverless-ohjelmistokehys

Cloudformation on erinomainen työkalu infrastruktuurin pystytykseen. Se ei kuitenkaan eliminoi kaikkia vaikeuksia infrastruktuurin päivittämisessä ja kehityksessä. Esimerkiksi jos Lambda-funktio on päivitetty, koko pinoa ei tarvitse tuhota ja luoda uudestaan. Riittää, että vain yksi tietty Lambda-funktio päivitetään. Päivityksen jälkeen funktio on aina vietävä Amazoniin ennen testausta ja ajoa. Jos funktioita on vähän, se ei ole

vaikeaa, mutta funktioiden määrän kasvaessa vienti alkaa jo tuntumaan liian työläältä. Tämän lisäksi jää vielä kaikkien palveluiden päivittäminen

- per kehitystiimi
- per alue (region)
- per lava (stage).

Serverless-ohjelmistokehyksestä on tässä apua, koska se piilottaa kehittäjältä vaaditut komentorivien suoritukset yhteen lyhyeen komentoon. Kehittäjän on vain laitettava konfiguraatiotiedostot valmiiksi, ja Serverless osaa itse hoitaa tarvittavat muutokset.

Lambda-palvelu ei aseta sovelluksen arkkitehtuurille tiukkoja vaatimuksia, mutta se on ehkä tärkein osa, joka on mietittävä ennen sovelluksen kehityksen alkua. Lambdaan sopivat sovelluksen pääarkkitehtuurit voidaan jakaa kolmeen eri tyyppiin, joita ovat

- monoliittinen pääarkkitehtuuri (monolithic)
- mikropalvelut (microservices)
- nanopalvelut (nanoservices).

Monoliittisella arkkitehtuurilla tarkoitetaan yleensä ohjelmaa, jossa kaikki tarvittavat koodit ovat samassa avaruudessa. Lambdassa se tarkoittaa sitä, että tehdään yksi iso Lambda-funktio, joka käsittelee kaikki API-kutsut.

Mikropalvelut-arkkitehtuuri tarkoittaa tässä tapauksessa montaa pienempää Lambda-funktiota per API-resurssi. Esimerkiksi API-resurssi

```
GET api/users
```

kutsuisi yhden Lambda-funktion, joka käsittelisi kaikki resurssiin liittyvät toiminnot (kuten luonti, haku jne.) päätepisteen avulla ja niiden http-verbit. Tässä arkkitehtuurissa kehittäjällä on isompi vapaus valita paremmin yhdelle resurssille sopivat työkalut.

Nanopalvelut-arkkitehtuurissa jokaista resurssin toimintoa varten luodaan oma Lambda-funktio. Edellä mainitun esimerkin mukaan seuraavat kolme API-kutsua käynnistäisivät kolme itsestään riippumatonta Lambda-funktiota:

```
GET api/users
GET api/users/maija.meikalainen
PUT api/users/maija.meikalainen
```

Koska Lambda-palvelussa saa (insinöörityön kirjoitusaikana) käyttää JavaScript-, Python- ja Java-ohjelmointikieliä, nanopalvelut-arkkitehtuuri tuntuu hyvältä, jos jokin yksittäisen resurssin toiminto eroaa huomattavasti kaikista muista. Esimerkkinä voi olla vanha JavaScript-ohjelmointikielellä kehitetty API-resurssi, jonka päätepisteeseen on pakko lisätä uusi toiminto, jolla saadaan tutkimuksen data eri diagrammeihin. Tässä tapauksessa voidaan käyttää nanopalvelut-arkkitehtuuria hyväksi ja lisätä API-resurssiin uusi päätepiste, joka käsittelee dataa Python-ohjelmointikielellä sen laajaa tieteellisten kirjastojen valikoimaa käyttäen. Tällöin yhteen API-resurssiin onnistuu integroida kaksi omiin tarpeisiin paremmin sopivaa työvälinettä.

KnoMe-sovelluksessa on käytetty mikropalvelut-arkkitehtuuria. KnoMe ei ole niin iso, että resurssissa tapahtuisi niin paljon bisnesslogiikkaa, että se olisi pakko pilkkoa vielä pienimmiksi osiksi. Toisaalta monoliittinenkaan arkkitehtuuri ei sopisi KnoMeen kovin hyvin, koska Serverless-palvelun integrointi olemassa olevaan sovellukseen tapahtuu yksi pala kerrallaan.

4 Integrointi KnoMe-sovellukseen

4.1 Käytetyt teknologiat

4.1.1 CoffeeScript, JavaScript, ES5/ES6, Node.js

JavaScript-ohjelmointikieli on tärkeä osa jokaista nykyaikaista web-sovellusta. KnoMessa kaikissa kolmessa osuudessa käytetään JavaScriptiä. JavaScript on vuonna 1995 kehitetty ohjelmointikieli, jonka avulla verkkosivuihin voidaan (käyttäjän selaimen tasolla) lisätä dynaamisuutta ja interaktiivisuutta. Viimeisin JavaScriptin standardi, jonka nimi on ECMAScript 6, tai lyhennettynä ES6, on hyväksytty vuonna

2015. Se lisää ohjelmointikieleen paljon kehitystä helpottavia ominaisuuksia, kuten nuolioperaatiot, let- ja const-avainsanat muuttujan initialisoinnissa, luokat (jotka todellisuudessa vain peittävät JavaScriptissa prototyyppipohjaisen periytymisen boilerplaten) sekä Set- ja Map-tietorakenteet. Valitettavasti kaikki selaimet (erityisesti vanhat) eivät tue ES6:a, joten kehittäjän pitää olla varovainen ES6:n uutuuksia käyttäessään. Onneksi kehittäjälle on tarjolla esimerkiksi Babel-niminen kirjasto, joka kääntää ES6-standardin rakenteet ES5-standardiin sopiviksi. Sen avulla kehittäjät voivat jo nyt aloittaa ongelmitta kehittämään käyttäen ES6-standardia. Myöhemmin, kun palvelimet ja selaimet tukevat ES6:a sataprosenttisesti, sovelluksessa voi helposti päästä eroon ES6:n rakenteiden kääntämisestä ES5:een ja käyttää uusinta standardia suoraan.

Vuonna 2009 Ryan Dahl esitteli Node.js-teknologian, joka perustuu Googlen kehittämään V8-tulkkiin. Node.js on tarkoitettu käytettäväksi palvelimella, ja se on asynkroninen ja käyttöjärjestelmäriippumaton teknologia. Vaikka ennenkin oli ollut erilaisia yrityksiä tuoda JavaScriptiä selaimesta palvelimelle, Node.js-tekniikan tulon jälkeen JavaScriptin suosio kasvoi entisestään, ja sitä alettiin käyttää palvelimella. EC2-instansseissa ja Lambda-palvelussa käytetty versio 4.3 tarjoaa tärkeimmät ES6:n ominaisuudet. Käytetyin niistä on Promise-tekniikka, jossa suoritus jää odottamaan, kunnes lupaus (promise) on toteutettu (resolved) tai hylätty (rejected). Promise-tekniikassa ei ole callback-tekniikkaan verrattuna mitään uutta. Promise vain helpottaa kehitystä tarjoamalla parempaa ja selkeämpää syntaksia, kuten then- ja catch-avainsanat, jotka voi yhdistää selkeään ketjuun. Se myös auttaa estämään sisäkkäisiä callbackeja, jotka vähentävät dramaattisesti koodin luettavuutta ja selkeyttä.

Viime aikoina JavaScript-maailmassa on tullut esille niin sanottuja source-to-source (lähdekoodista lähdekoodiksi) -kääntäjiä. Source-to-source-kääntäjä ottaa lähdekoodin ja kääntää sen toisen kielen lähdekoodiksi. Tällä tavalla on mahdollista korvata kokonaan, muuttaa tai helpottaa käännetyksi tulevan ohjelmointikielen syntaksia tai rakenteita. Source-to-source-kääntäjien käyttäminen myös tarjoaa mahdollisuuden lisätä alkuperäiseen ohjelmointikieleen siitä puuttuvia ominaisuuksia. Ehkä kaikista tunnetuimmat JavaScriptiksi käännettävät ohjelmointikielet ovat CoffeeScript ja TypeScript.

JavaScriptissä on dynaaminen tyypitys, joka voi isolla koodimäärällä olla harhaanjohtava ja sekava. Tämän lisäksi kehittäjän käyttämät kehitysympäristöt, jotka analysoivat lähdekoodia, eivät aina osaa hyvin vihjata virheistä ja täydentää muuttujia.

Esimerkiksi yksi parhaista työkaluista, Intellij IDEA, tarjoaa JavaScriptissä yleensä melkein kaikki, usein väärätkin funktiot, eikä vain tiettyyn luokkaan tai objektiin liittyviä funktioita. TypeScript yrittää ratkaista tämän ongelman tarjoamalla lähdekoodissa tyyppejä, jotka ohjelmoija voi tarvittaessa määritellä. TypeScriptin tyypitys ei tietenkään tee JavaScriptistä staattisen tyypityksen ohjelmointikieltä, vaan se vain tarjoaa hyvän mahdollisuuden napata osan virheistä jo ennen kompilointia. Samalla kehitysympäristöt ymmärtävät paremmin koodia. TypeScript tarjoaa paljon muutakin, kuten luokkia, rajapintoja, luetteloja jne. [7]

Toisaalta CoffeeScript on pienempi ratkaisu, joka tarjoaa vain kirjoittamissyntaksia helpottavia rakenteita. Se tarjoaa esimerkiksi avainsanoja kuten ”is” (===), ”isnt” (!==), ”and” (&&) ja ”or” (||), automaattisen viimeisen rivin palautuksen, optionaalisia sulkuja, function-, this- ja bind-avainsanoja helpottavia rakenteita jne. CoffeeScriptiä käytetään KnoMessa sekä selaimessa että vanhalla palvelinpuolella. Seuraava kaappaus antaa pienen kuvan siitä, miltä CoffeeScript näyttää. Kuvasta 4 voidaan nähdä, miten funktioita ja for-silmukka määritetään. Lisäksi if-vertailussa ja for-silmukassa sulut eivät ole pakollisia, mikä voi olla sekavaa, jos kehittäjä näkee CoffeeScriptin ensimmäisen kerran.

```
updateSkillInterestText = (skill) ->
    skill.interest = parseInt(skill.interest) || 100
    for interest in skillInterests
        do (interest) ->
            if interest.range_min <= skill.interest and interest.range_max >= skill.interest
                skill.interestText = interest.description_en
```

Kuva 4. CoffeeScript muistuttaa aika paljon JavaScriptiä. Sen tarkoitus on vain helpottaa kehittämistä ja eliminoida turhat avainsanat, eikä se lisää JavaScriptiin mitään uutta.

KnoMe-sovelluksen analysoinnin perusteella totesimme, että CoffeeScript tuo usein ylimääräistä työtä, koska usein on pakko katsoa, millaiseksi JavaScriptiksi se on kääntänyt, ja jättää koodi sovellukseen vain, jos käännetty JavaScript on oikeata muotoa. Muutaman kerran törmäsimme ohjelmointivirheisiin CoffeeScriptin automaattisen viimeisen rivin palautuksen takia. Totesimme pääkehittäjän kanssa, että CoffeeScript on nykyisissä projekteissa turha vaihtoehto.

4.1.2 Gulp-tehtävänsuorittaja

Serverless auttaa vaiheen (stage) vientiä Amazon AWSiin huomattavasti, mutta silti sen lisäksi tarvittiin pieniä apuskriptejä. Tarkoituksena oli helpottaa ja automatisoida erilaisia tehtäviä, joita tarvitaan ennen vientiä tai sen jälkeen. Esimerkkinä voi olla S3-levyn tyhjennys ennen vientiä ja tietokannan täyttäminen testidatalla viennin jälkeen. Myöhemmin sovellukseen lisättiin apuskriptejä tietokannan ja S3-levyn varmuuskopiointia varten, jota Amazon ajaa joka päivä. Apuskriptit oli päätetty kehittää JavaScriptillä tehdyn Gulp-tehtävänsuorittajan avulla, koska silloin se osaisi käyttää sovelluksessa jo olemassa olevia funktioita ja luokkia.

Tehtävänsuorittaja on apuväline, jota käytetään kehityksen nopeuttamiseksi. Se ajaa valmiiksi määritettyjä tehtäviä, jotka ovat JavaScript-ohjelmointikielen funktioina. Gulp-tehtävänsuorittajaa varten on paljon erilaisia lisäyksiä [8] eri tarkoituksiin. Niitä löytyy mm.

- JavaScript-koodin minimointia varten (gulp-uglify)
- HTML- ja CSS-tiedostojen minimointia varten
- Muutosten jälkeen palvelimen uudelleen käynnistämistä varten (gulp-watch)
- Esikääntämistä varten (gulp-less)

Paikallisesti käytämme vain itse tehtyjä skriptejä. Kuvassa 5 näkyy osa Gulp-skriptistä Serverless-sovelluksen initialisointia varten. Kuvassa näkyy, miten Gulp osaa luoda uuden prosessin (spawnSync), ja miten se kutsuu apukirjaston funktion S3-levyllä olevien tiedostojen tuhoamista varten. Jos tämä suoritetaan onnistuneesti, Gulp lopettaa nykyisen skriptin ja siirtyy suorittamaan seuraavan (cb()). Muussa tapauksessa se tulostaa virheen ja lopettaa ajon (cb(err)).

```
35    // Remove files from attachments S3 bucket before re-creating CloudFormation stack
36    gutil.log("Clearing S3 attachments bucket for CloudFormation handling");
37    S3.clearS3s(clearBuckets,stage).then( (result) => {
38
39        // Remove existing CloudFormation stack, that we can recreate it again with new resources
40        var slsCFRemove = spawnSync('sls', ['resources', 'remove', '-s', stage, '-r', 'eu-west-1']);
41        gutil.log(`${slsCFRemove.stdout}`);
42        gutil.log("Waiting CloudFormation removal for 70 seconds");
43        spawnSync('sleep', ['70']);
44
45        gutil.log("Re-deploying CloudFormation (~ 3 minutes)");
46        var slsCFDeploy = spawnSync('sls', ['resources', 'deploy', '-s', stage, '-r', 'eu-west-1']);
47        gutil.log(`${slsCFDeploy.stdout}`);
48
49        cb();
50
51    }).catch( (err) => {
52        cb(err);
53    });
```

Kuva 5. Kaappaus Gulp-apuskriptistä, joka initialisoi uuden Serverless-sovelluksen ja poistaa S3-levyllä olevat tiedostot.

4.1.3 Jade

Jade [9] on kirjasto, joka kääntää tiivistä, HTML:lää muistuttavaa syntaksia oikeaksi ja täydeksi HTML-syntaksiksi. Jaden syntaksi eroaa tavallisesta HTML-kuvauksesta aika vähän, joten ohjelmoijan ei tarvitse kovin paljon opiskella uutta, jotta hän voisi aloittaa kehittämään Jaden syntaksilla. Jade tarjoaa mm. seuraavat hyödyt, joita käytettiin KnoMessa:

Jadessa ei tarvitse käyttää sulkuelementtejä, niin kuin HTML-syntaksissa. Sisennyksellä luodaan sisäkkäisiä elementtejä. Include-määritelmällä kuvaukseen voidaan helposti lisätä olemassa oleva tiedosto, joten koodin uudelleenkäytettävyys nousee hyvälle tasolle. Luokat luetellaan peräkkäin pisteellä erotettuina tagin nimen jälkeen, muut tarpeelliset sanat tulevat sulkuihin. Kuvassa 6 näkyvät kaikki luetellut asiat.

```
2   section(ng-controller="CustomersProjectCreateController")
3       .row(ng-class="{fullscreen: fullScreen, paddingSides30: fullScreen}")
4           .col.col-xs-12.paddingTop10
5               include ../misc/fullscreen_button
6
7               form#projectCreateForm(novalidate, name="projectcreateform")
8                   .row
9                       .col-sm.col-xs-12#project-create-project-name
10                          .row
11                              .col-xs-12
12                                  label Project Name:
13                              .col-xs-12
14                                  input.form-control.input-sm(id="project_name",
15                                                              name="projectName",
16                                                              type="text",
17                                                              minlength="1",
18                                                              ng-model="project.project_name",
19                                                              ng-maxlength="100",
20                                                              required)
21                              .col-xs-12.has-error#nameErrorRequired(ng-show="
                                projectcreateform.projectName.$error.required")
22                                  span.help-block Project name is required
```

Kuva 6. Kaappaus Jade-syntaksista. Kuvassa näkyy, että Jaden syntaksi muistuttaa HTML-syntaksia. Ainoa silmiinpistävä ero on pisteellä erotettuina luetellut luokat.

4.1.4 Chef

Chefin avulla KnoMen vanhaa palvelinta varten luodaan uusi Amazon EC2 -instanssi, otetaan yhteyttä siihen ja asennetaan kaikki tarvittavat kirjastot ja moduulit.

Chef on Configuration management tool -ohjelma, jossa käyttäjä tekee niin sanottuja reseptejä, jotka kuvaavat tavoitetilaa kohteessa. Reseptissä käyttäjä kuvaa Chefin syntaksilla, mitä se haluaa Chefin tekevän. Chef voi esimerkiksi ajaa komennon, luoda tiedoston, muuttaa tiedoston oikeudet, asentaa kirjastoja, hakea koodia jostakin paikasta, luoda pilvessä uuden instanssin, hakea salasanat kryptatusta tiedostosta ja paljon muuta. Kuvassa 7 näkyy yksi Chefin resepti. Aluksi tulevat muuttujat ja niiden arvot, sitten Chef ottaa mallitiedoston source-kohdan polusta, sijoittaa muuttujat ja kopioi lopputuloksen template-kohdassa mainittuun polkuun. Kun Chef käynnistetään kehitysympäristössä, voimme tällöin automaattisesti kytkeä pois päältä Google Tag Manager- tagien hallintajärjestelmän, eikä tulevan kehittäjän tarvitse huolehtia siitä. Sitten Google Tag Manager laitetaan päälle tuotantopalvelimella.

```
73  ##
74  # Copy static Google Tag Manager
75  ##
76  google_tag_manager_account_id = "[redacted]"
77  google_tag_manager_use_gtm = true
78  google_tag_manager_use_gtm = false if node.chef_environment == "development"
79
80  template "#{node['knome']['src_home']}/client/views/partials/misc/google_tag_manager.jade" do
81    variables(
82      :gtmAccountId => google_tag_manager_account_id,
83      :useGtm => google_tag_manager_use_gtm
84    )
85    source 'frontend-static/google-tag-manager.jade.erb'
86    user "knome"
87  end
```

Kuva 7. Chefin ”resepti”, joka sijoittaa arvot muuttujiin ja kopioi lopputuloksen template-kohtaan laitettuun polkuun

4.2 KnoMe-sovellus

KnoMe on yrityksen sisäiseen käyttöön kehitetty sovellus. Alun perin sovelluksen ideana oli toimia yrityksen työntekijöiden ansioluettelopankkina. Kun uusi työntekijä otetaan yritykseen työhön, häntä varten luodaan uusi profiili. Työntekijä pystyy kirjautumaan sisään niin sanotuilla Active Directory -tunnuksilla, joita it-tuki ylläpitää. Active Directory on Microsoftin käyttäjätietokanta ja hakemistopalvelu [10].

Käyttäjien profiilien ja ansioluetteloiden lisäksi sivulta löytyy erilaisia raportteja. Raportit-välilehdeltä löytyvät esimerkiksi muutokset henkilöstössä ja sen määrässä, haku taidon perusteella, diagrammi työntekijöiden iästä ja työntekijöiden saamat sertifikaatit. Sovellus ei kata kaikkia alueita, mutta toimii hyvänä työkaluna henkilöstöosastolle.

Sovellus on saanut kehitysimpulssin viime vuonna. Noin kaksi henkilöä alkoi siitä lähtien kehittää sitä täyspäiväisesti. Olin KnoMen pääkehittäjän parina noin 16.3.2016 ja 3.8.2016 välisenä aikana. Silloin Lambda-palvelun integrointi oli korkeassa prioriteetissa ja ainoana tehtävänä muita pieniä työtehtäviä ja päivityksiä lukuun ottamatta.

KnoMe ei ole sovelluksena kovin iso. Eräs tapa hahmottaa sovelluksen suuruus on laskea koodirivien määrä. Taulukko 1 kertoo JavaScript-ohjelmointikielen rivien määrästä kolmessa sovelluksen osiossa:

- Front end, joka on käyttäjälle tarkoitettu osio. Se on koodi, joka toimii päätelaitteessa, eli esimerkiksi selaimessa. Se vastaa tietyn sivun päivittämisestä ja renderöinnistä, datan näyttämisestä ja interaktiivisesta käyttöliittymästä.

- Back end, joka toimii palvelimena. Se ohjaa tulevat kutsut, palauttaa dataa tietokannasta tai Amazonin kutsuista, generoi pdf-tiedoston tulostamista varten, luo ja päivittää dataa.

- Amazon Lambda on pilvessä oleva sovellus.

Taulukko 1. Taulukko kertoo sovelluksen osion koodirivien määrästä ja tarkoituksesta.

Osio	Koodirivien määrä	Tarkoitus (joitakin esimerkkejä)
Back end	6 500	Web-kutsujen ohjaus, datan tallentaminen, haku ja poisto, pdf-tiedoston generointi.
Front end	4 500	Selaimessa oleva käyttöliittymä ja siihen liittyvät kaikki vuorovaikutukset
Amazon	10 000	Amazon AWS:ssä toimiva sovellus

Koodirivien määrä ei välttämättä korreloi suoraan sovelluksen vaikeuden kanssa. Siihen vaikuttavat monet sovelluksessa käytetyt asiat. Suurin vaikutus on käytetyllä ohjelmointikielellä, sillä joitakin funktionaaliset ohjelmointikielet, kuten esimerkiksi Scala, tarjoavat mahdollisuuden todella suppeaan syntaksiin ja funktionaalisiin rakenteisiin. Ja vaikka koodirivien määrä voi silloin olla pienempi, funktionaalisessa ohjelmoinnissa koodissa voi olla paljon enemmän toiminnallisuutta. Toiseksi vaikuttavin asia on koodityyli. Esimerkiksi if-lausekkeen tyyli ei muuta sovelluksen tai logiikan vaikeutta, mutta voi tyylistä riippuen hyvin vaikuttaa koodirivien määrään.

```
if {
…
} else {
…
}
```

Koodiesimerkki 10. Joissakin projekteissa käytetty tyyli kirjoittaa if-lauseketta. Tässä if-lauseke on vain kolme koodiriviä neljän (jos else-osa tulee omalle rivilleen) tai kuuden (jos hakasulutkin laitetaan omille riveilleen, nyt käytetään hyvin harvoin) sijaan.

Vanhan sovelluksen analysointi

Vanha sovellus on tavallinen REST-palvelin, joka ei tee raskaita laskuoperaatioita, lukuun ottamatta pdf-tulostusta ja tag-pilven generointia, johon tarvitaan oma ohjelma, R- ja Java-ohjelmointikieliä sekä tarvittavat riippuvuudet asennettuina. Suurin osa palvelimien operaatioista on CRUD-operaatioita. Tämän takia päätimme käyttää yhtä pientä palvelinta pdf- ja tag-pilven generointia varten ja suorittaa loput toiminnot Lambda-funktioiden kautta. Näimme siinä ainakin seuraavat hyödyt:

- EC2:n instanssin tai palvelimen kaatuminen. On hyvin epätodennäköistä, että itse EC2-instanssi kaatuu, koska Amazon takaa vähintään 99,95 %:n käytettävyysajan kuukaudessa [11]. Todennäköisempi on tilanne, jossa ohjelmoija jättää jonkin virheen tarkistamatta, minkä takia palvelin kaatuu, ja sivuun ei enää ole mahdollista ottaa yhteyttä. Vanhassa sovelluksessa kaikki toiminnot riippuvat palvelimesta, ja sen kaatuminen tarkoittaa koko KnoMen menemistä pois päältä. Uutta tapaa käytettäessä palvelimen tai jopa koko EC2-instanssin hajoaminen tarkoittaa vain sitä, että käyttäjä ei pääse tiettyihin palveluihin, kuten raportteihin tai pdf-generointiin, jotka eivät ole niin kriittisiä toimintoja. Myös fataalivirhe Lambda-funktiossa estää vain yhden API-resurssin käytön, eikä kaikkea.

- Laskutus. Lambda on huomattavasti edullisempi, koska siinä laskutetaan vain käyttökerroista ja RAM-muistin allokaatiosta. EC2-instanssissa laskutetaan jokaisesta tunnista, joten kevyin EC2-instanssi + Lambda-funktio säästäisi yritykselle rahaa. Isossa sovelluksessa laskutustavan vaikutus menoihin voi olla paljon isompi.

- Skaalautuvuus. Vanhaa sovellusta ei ollut mahdollista skaalauttaa helposti moniin palvelimiin. Lambda-palvelussa Amazon ottaa skaalautuvuuden itse hoidettavakseen, joten vaikka käyttäjämäärä kasvaisi sadasta uniikkikäyttäjästä sataan tuhanteen uniikkikäyttäjään, se ei vaikuttaisi sovelluksen suorituskykyyn ja ylläpitotöihin niin kriittisesti kuin jos koko sovellus olisi yhdellä palvelimella.

Vaikka edellä mainitut syyt ovat hyvin tärkeitä ja jo pelkästään niiden takia olisi hyvä ajatella siirtymistä Lambda-palveluun, KnoMen yhteydessä ne eivät olleet kaikista tärkeimpiä. Tärkeimpänä syynä oli kokeilla, analysoida tuloksia ja rakentaa koodipohja tulevaisuutta varten, jolloin yritys voi käyttää koodia ja kokemusta asiakasprojekteissa.

Ennen integroinnin alkua KnoMe oli hyvin käyttökelpoisessa vaiheessa. Se helpotti asiaa, koska pystyimme alussa ajattelemaan rauhassa erilaisia aloitustapoja, eikä meidän tarvinnut korjata olemassa olevaa toiminnallisuutta. Totesimme silloin, että integrointi API-kutsujen kautta olisi ainoa ja paras vaihtoehto. Tällä hetkellä vanha palvelin vain ”piilottaa” oikeat API-kutsut käyttäjältä tarjoamalla omat osoitteensa, jotka ohjaavat melkein suoraan Amazon API Gateway -palvelun osoitteisiin. Syynä tällaiseen turhaan systeemiin on Active Directory -palvelun auktorisointi ja sen yhteensopivuus Amazon Cognito -palvelun kanssa, joka tarjoaa mahdollisuuden auktorisoida käyttäjiä. Integroinnin aikana emme löytäneet parempaa vaihtoehtoa kuin asettaa Cognito-palvelun käyttöönotto tavoitteeksi tulevaisuuteen.

Integrointi voidaan jakaa kolmeen eri osaan:

1. Alku
2. Sovelluksen lopullinen rakenne
3. S3-palvelun lisäys.

Alkuvaihe

Integrointi alkoi, kun KnoMeen tuli tarve kehittää asiakas-entiteetti, jonka tarkoituksena on kuvastaa yrityksen asiakasta. Aloitimme tutustumalla Serverless-ohjelmistokehykseen, sen toiminnollisuuteen ja Amazon AWS-palveluihin. Loimme testiprojekteja ja tutkimme, miten se vie paikalliset koodit ja mallitiedostot Amazoniin.

Havaitsimme, että vaihe on initialisoitava, ennen kuin siihen viedään Lambda-funktiot, laukaisut ja API Gateway -resurssit. Kehityksen aikana tulee usein tarve poistaa vaihe kokonaan ja luoda se uudestaan, joten otimme käyttöön Gulp-suorittajan ja teimme viemistä varten valmiita Gulp-tehtäviä kehittäjän ajan säästämiseksi. Alussa teimme

vain muutamaa tehtävää, joiden tarkoituksena oli luoda vaihe, odottaa noin 70 sekuntia ja viedä sitten kaikki palvelut. Tämän jälkeen havaitsimme, että testidataa on joka tapauksessa vietävä tietokantaan, joten teimme myös tätä varten tehtävän, jota Gulp ei poikkeuksellisesti aja, jos vaiheen nimi on ”production” (tuotanto). Olimme myös alussa päättäneet, että testit ovat tärkein osa uutta sovellusta, joten yritimme saada mahdollisimman ison testikattavuuden. Jokaisen resurssin päätepisteelle tuli oma, mahdollisimman iso määrä testejä, jotka Gulp ajaa.

DynamoDB:n kanssa ei ollut paljon ongelmia, koska entiteettien rakenne oli selkeä. Asiakas-entiteetissä tärkeimmät kentät ovat nimi, kuvaus, web-sivu, GUID, viimeksi päivitetty aikaleima ja työntekijöiden määrä. KnoMe sallii kaikkien muiden kenttien paitsi nimi-kentän jättämisen tyhjäksi. Heti alussa huomasimme DynamoDB:n ominaisuuden, että se ei hyväksy tyhjiä kenttiä. Tämän vuoksi teimme vähän ylimääräistä työtä kehittääksemme jaettuja funktioita, jotka poistavat tyhjät kentät ennen tallennusta. Samalla toteutimme pienen apukirjaston tallennusta varten. Vaikka SDK tarjoaa jo funktioita tallennusta ja hakua varten, teimme Amazonin funktioita käyttäen oman Promise-tekniikkaan perustuvan implementaationsa.

API on tehty CRUD-menetelmän mukaan, eli HTTP GET -pyyntö vastaa datan hausta, POST-pyyntö entiteetin luomisesta, PUT-pyyntö entiteetin päivityksestä ja DELETE-pyyntö entiteetin poistosta. Todellisuudessa poisto-operaatio eroaa vähän oikeasta poistosta. Se vain merkitsee entiteetin poistetuksi, minkä jälkeen entiteettiä ei enää haeta, päivitetä eikä näytetä.

Sovelluksen lopullinen rakenne

Tässä vaiheessa sovellus oli saanut lopullisen rakenteensa ja arkkitehtuurinsa. Syynä arkkitehtuurin muutokseen oli tiedon ja kokemuksen (Serverless:istä ja Amazonista) puute. Meillä ei ollut, emmekä löytäneet kovin paljon tietoa pilvessä olevan sovelluksen parhaista käytännöistä. Aiemmin valittu arkkitehtuuri muuttui liian vaikeaksi, eikä enää helpottanut kehitystä. Koodi tuntui liian ”yhteenliimatulta”. Koodin liimaus on Lambda-funktioiden filosofian vastaista. Tämän filosofian mukaan Lambdan tulee olla nopea, pieni ja suhteellisen riippumaton muista. Vanhassa arkkitehtuurissa oli isoja, kaikkien käytettävissä olevia apufunktioita, kuten tulevan ja ennen tallentamista olevan tiedon validointi. Uudessa arkkitehtuurissa otimme käyttöön validointikirjaston ja teimme tarvittavat apufunktiot jokaisen Lambda-funktion kansioon. Tällöin koodia ja jossakin paikassa jopa DRY-menetelmän (Don’t Repeat Yourself -menetelmä, jossa koodia

pyritään uudelleenkäyttämään mahdollisimman paljon) vastaista rakennetta tuli enemmän, mutta jokainen Lambda-funktio tuli riippumattomammaksi muista.

Validointia ei tehdä vain tulevalle tiedolle. Entiteetissä on sellaisia kenttiä, joita vain Lambda-funktio itse saa päivittää. Esimerkkinä voivat olla entiteetin päivittäjän tai luojan id:t ja aikaleimat. Vaikka näitä kenttiä voi muuttaa käsin selaimessa kehittäjän työkalujen avulla, ja on mahdollista lähettää väärät aikaleimat, Lambda-funktiossa oleva validointi ei mahdollista sellaisen datan ylikirjoittamista.

Tässä vaiheessa aloitimme kehittämään projekti-entiteettiä asiakas-entiteetille. Koska projekti- ja asiakas-entiteetit ovat vahvasti liitetyt toisiinsa, käyttöliittymässä halutaan yleensä nähdä molempien tiedot samaan aikaan. DynamoDB ei valitettavasti tarjoaa mahdollisuutta liittää tai yhdistää tietoja useista tauluista, joten se piti tehdä KnoMen käyttöliittymässä eli käyttäjän selaimessa. Tämä lisäsi vähän ongelmia, koska vaikka entiteettien määrä KnoMessa ei ole iso, niiden yhdistäminen, haku ja kuvauskielen näyttäminen kaikki mahdolliset tilanteet huomioon ottaen muuttui nopeasti hankalaksi.

Tässä vaiheessa sovelluksen arkkitehtuuri sai lopullisen muotonsa, eikä enää muuttunut.

S3-palvelun lisäys

Seuraavassa vaiheessa lisäsimme S3-palvelun, jota varten teimme apufunktioita ja uusia Gulp-tehtäviä. Olimme ottaneet S3-palvelun käyttöön kuvien ja liitteiden tallentamisen lisäksi myös varmuuskopiointia varten. Koko DynamoDB-tietokannan ja KnoMeen tallennettujen liitteiden varmuuskopiointi tehdään Lambda-funktiolla joka päivä. Gulp-tehtävän avulla varmuuskopiointi voidaan ajaa käsin tai tallentaa tietokoneen levylle. Kehitimme myös Gulp-tehtävän, joka palauttaa varmuuskopioinnista dataa DynamoDB-tietokantaan tai liitteet S3-palveluun. Gulp-tehtävä käyttää paikallisesti käyttöön ladattua Amazon SDK -kirjastoa, joka tarjoaa tarvittavat funktiot tallentamista, poistoa ja palauttamista varten.

Amazon tarjoaa erilaisia tapoja ladata tiedostoja S3-palveluun. KnoMessa päätimme tehdä tulevan kutsun validoinnin vanhalla palvelimella, joka lataa tiedoston palveluun itse. Vaikka se on selvää tuplatyötä, eli käyttäjä lataa vanhalle palvelimelle, ja vanha palvelin lataa sitten S3-palveluun, syynä oli Cognito-palvelun konfiguroimattomuus ja auktorisoinnin puute. Yksi vaihtoehto oli antaa pääsy käyttäjän ladattuihin ja S3-

palvelussa oleviin tiedostoihin julkisilla linkeillä, jotka ovat pitkiä, vaikeasti arvattavia merkkijonoja. Tämä vaihtoehto katsottiin kuitenkin liian vaaralliseksi, koska linkki on mahdollista ottaa talteen ja antaa vieraille. Tallentaminen S3-palveluun toimii samalla tavalla. Vanha palvelin tarkistaa oikeuden liitteen tallentamiseksi ja antaa generoidun linkin, joka toimii 60 sekuntia. Lambda-funktio generoi joka kerta uuden linkin, jonka vanha palvelin välittää käyttäjän selaimelle. Käyttäjä saa tallentaa tiedoston tällä aikavälillä. Vaikka teoriassa linkkiä saa käyttää kuka tahansa, sitä ei katsottu liian vaaralliseksi, koska 60 sekuntia elävän, yhden tiedoston tallentamista varten generoidun linkin vuoto ei haittaisi KnoMen toimintaa.

S3-palvelun käyttöönotossa tutustuimme toiseen mahdollisuuteen laukaista Lambda-funktio. Ennen sitä kaikki Lambda-funktiot laukaistiin API Gateway -palvelusta tulevan API-kutsun jälkeen. Tässä vaiheessa lisäsimme laukaisun tietyllä ajankohdalla, eli varmuuskopiointi klo 3:00 ja tiedoston S3-levylle lataus tai S3-levyltä poisto. Laukaisuja on mahdollista laittaa melkein jokaiselle Amazonin palvelulle, tämä mahdollisuus on hyvin vahvasti integroitu Amazoniin. Muutama Lambda-funktioista laukaistaan, kun tiedosto on ladattu tai poistettu S3-palvelusta. Amazon laukaisee automaattisesti Lambda-funktion tietyillä parametreilla, joista voi saada selville tiedoston nimen, levyn nimen, aikaleimat jne. S3-palvelussa olevan tiedoston polku voi olla esimerkiksi seuraavaa muotoa:

bucketName/attachments/customers/:customerId/:fileName

Polusta näkyy siis heti tiedoston nimi, ja mitä entiteettiä Lambda-funktion pitää päivittää (asiakas-entiteetti ja asiakkaan id). Funktio ottaa laukaisun jälkeen yhteyttä tietokantaan ja päivittää tarvitun entiteetin.

Näiden vaiheiden jälkeen en enää osallistunut KnoMen kehitykseen.

4.3 Vaikeudet integroinnissa

Integroinnin aikana havaitsimme muutamia vaikeuksia. Ehkä kaikista isoin ja aikaa vievin vaikeus on mahdottomuus ajaa Lambda-funktioita paikallisesti. Pienen muutoksen jälkeen on aina ladattava funktio Amazoniin, ja vasta sen jälkeen voi ajaa testejä tai kokeilla muutosta käsin. Tämän lisäksi lokit löytyvät vain CloudWatch-palvelusta, ja ne pitää laittaa päälle, jos niitä ei ole laitettu päälle automaattisesti.

Koska funktioita ei voi ajaa paikallisesti, ei ole mahdollista asettaa pysäytyspistettä (breakpoint) koodin riviksi ja analysoida koodin ajamisen kulkua (debuggata) rivi riviltä. Vaikka se ei ole kriittinen vaikeus, se vie huomattavasti aikaa kehittäjältä, joka ei ole tutustunut ohjelmointikieleen tai -kehykseen.

Päätimme käyttää integroinnissa JavaScript-ohjelmointikieltä, joten sen erikoispiirteet oli tiedettävä ohjelmointivirheiden välttämiseksi. Esimerkkinä voi olla Object.assign(target, …sources) -funktio, jonka tarkoituksena on kopioida objektin ominaisuudet (properties) eli kloonata objekti. Etukäteen on tiedettävä, että se ei tee syväkloonausta eli kopioi objektissa olevien objektien ominaisuuksia. Jos objektissa olevan objektin ominaisuuksia muutetaan kopioinnin jälkeen, ne muuttuvat kaikissa muuttujissa, joihin muutettu objekti on kloonattu. Tämä lisäsi virheiden etsimiseen kuluvaa aikaa, koska emme tienneet tästä funktion ominaisuudesta etukäteen.

Serverless-ohjelmistokehys helpotti kehitystä, mutta joskus se ilmoitti hyödyttömiä virheitä, joista emme saaneet selvää, mihin ne liittyivät. Esimerkiksi seuraava virhe oli jossakin vaiheessa osana sovellustamme:

…TypeError: e.message.split is not a function…

Vaikka virhe sisältää muut jäljet, virheestä ei saa ollenkaan selvää, mitä on mennyt väärin ja missä. Teksti vain ilmoittaa, että Serverlessin sisällä tapahtuneessa virheessä ei ole viestiä, Serverless ei kerro sen enempää. Kävi ilmi, että Amazonissa oli kesken stackin poisto, joten Serverless ei luonut uutta stackia, ja kun se yritti viedä siihen tavaraa, Serverlessissa tapahtui virhe.

Nämä ovat ehkä isoimmat vaikeudet, joihin törmäsimme kehityksen aikana.

5 Yhteenveto

Tässä insinöörityössä olemme tutustuneet Amazon AWS -pilvipalveluihin, KnoMe-sovellukseen, siinä käytettyihin nykyaikaisiin teknologioihin ja pilvipalveluiden integrointiin.

Mielestäni suoritimme integroinnin onnistuneesti lukuun ottamatta ongelmaa Active Directory -palvelun kanssa. JavaScript ohjelmointikielenä ei tuonut kovin paljon

ongelmia. Vaikka se vaatii JavaScriptin erityispiirteiden tuntemista, ne on helppo oppia. Amazon Lambda tukee Node.js:n versiota 4.3, joka tarjoaa monta kehitystä helpottavaa ES6:n ominaisuutta. Tämän takia siirsimme kaikki Lambda-funktiot käyttämään Node.js 4.3 –versiota, mikä helpotti kehitystä ja siisti koodia. CoffeeScriptiä käytetään mielestäni aivan turhaan, koska se lisää rasitusta kehitykseen ja vaatii kääntäjän palvelimella, selaimessa ja testejä varten. Muuttaisin mielelläni CoffeeScriptin tavalliseksi JavaScriptiksi ja pääsisin siitä eroon.

Sinä aikana, kun kehitin sovellusta, teimme integrointia suunnilleen suunnitelman mukaan. Projektiin tuloni aikana Serverlessiä oli vasta aloitettu käyttämään. Pikkuhiljaa oli luotu asiakas-entiteetti, jota käytetään nyt aktiivisesti KnoMessa. Sen jälkeen lisäsimme KnoMeen asiakasprojekteja ja mahdollisuuden ”tykätä” asiakkaista, henkilökohtaisia projekteja ja mahdollisuuden ladata asiakkaiden logoja tai liitteitä. Näiden asioiden lisäksi päivitimme vanhaa palvelinta ja selaimessa olevaa koodia. Meitä kehittäjiä oli vain kaksi, joten mielestäni integroinnin tahti oli hyvä.

Serverless-ohjelmistokehys ja niin sanottu event-triggered computation ovat loistavia tapoja kehittää tulevaisuuden sovelluksia. Vaikken suosittelisi käyttämään Serverless:iä asiakasprojekteissa sen ollessa 0.56 alpha -versiossa, näen sen oikeana työkaluna, kun se kypsyy ja siitä tulee 1.0 -versio.

En osaa arvioida, minkä paikan Serverless ja event-triggered computation löytää markinnoilla, mutta jo nyt on selvää, että muuttamalla sovelluksen mikropalvelut-arkkitehtuuriksi ja Lambda-palveluun sopivaksi muodoksi yritys voisi säästää hyvän osan menoistaan.

Lähteet

1. Virtualisointi. Verkkodokumentti < https://fi.wikipedia.org/wiki/Virtualisointi >. Luettu 10.7.2016.
2. Amazon SDKs < https://aws.amazon.com/tools/#sdk >. Luettu 20.7.2016.
3. TIOBE. Popularity of programming languages < http://www.tiobe.com/tiobe_index >. Luettu 20.7.2016.
4. Microsoft Azure < https://azure.microsoft.com/en-us/ >. Luettu 25.7.2016.
5. Amazon AWS case studies < https://aws.amazon.com/solutions/case-studies/ >. Luettu 26.7.2017.
6. Amazon Cloudformation. DynamoDB:Table documentation. < http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-dynamodb-table.html >. Luettu 1.8.2016.
7. TypeScript < https://www.typescriptlang.org/ >. Luettu 22.8.2016.
8. Essential Gulp Plugins. < https://github.com/Pestov/essential-gulp-plugins >. Luettu 18.8.2016.
9. Jade Template Engine. < http://jade-lang.com/ >. Luettu 19.8.2016.
10. Active Directory. < https://fi.wikipedia.org/wiki/Active_Directory >. Luettu 9.8.2016.
11. Amazon EC2 Service Level Agreement. < https://aws.amazon.com/ec2/sla/ >. Luettu 18.8.2016.