SAVONIA
ammattikorkeakoulu

OPINNÄYTETYÖ – TIETOTEKNIIKAN TUTKINTO-OHJELMA
TEKNIIKAN JA LIIKENTEEN ALA

OHJELMISTOKEHITYS SERVERLESS-ARKKITEHTUURISSA

ARKKITEHTUURIN HYÖDYT JA HAASTEET

TEKIJÄ Daniel Mäkinen

Koulutusala Tekniikan ja Liikenteen ala	
Tutkinto-ohjelma Tietotekniikan tutkinto-ohjelma	
Työn tekijä(t) Daniel Mäkinen	
Työn nimi Ohjelmistokehitys serverless-arkkitehtuurissa	
Päiväys 24.03.2024	Sivumäärä/Liitteet 26 / 0
Toimeksiantaja/Yhteistyökumppani(t) Vincit Oyj - Tampereen Särkänniemi Oy	

Tiivistelmä

Opinnäytetyön tarkoituksena oli dokumentoida ohjelmistokehitysprosessia serverless-arkkitehtuuriin perustuvassa järjestelmässä, samalla kun työstin siihen uutta toimintoa. Tavoitteena oli kuvailla, miten kehitysprosessit eroavat työstäessä kyseistä arkkitehtuuria verrattuna perinteisempiin palvelinpohjaisiin arkkitehtuureihin, eli mitkä prosessit suoraviivaistuvat ja mitä erilaisia haasteita serverless-arkkitehtuuri tarjoaa. Työstettävä projekti perustui paljon Amazon Web Serviceissä isännöityihin palveluihin ja siksi tämä kirjoitelmakin keskittyy paljon heidän tarjontaansa.

Ensimmäiseksi käydään läpi, miten ohjelmistoarkkitehtuuri määritellään ja mitä serverless-arkkitehtuuri tarkoittaa tässä suhteessa. Sitten kuvaillaan työstettävän järjestelmän rakennetta, jonka jälkeen kerrotaan toiminnon kehityksen yhteydessä vastaan tulleista työnkulun vaiheista. Lopuksi vielä tiivistän ajatuksiani siitä, mitä yleisiä teemoja serverless-arkkitehtuurin kehityksessä esiintyi ja millaisia ajatuksia arkkitehtuurista jäi käteen toiminnon valmistuttua.

Avainsanat

pilvipalvelut, ohjelmistokehitys, serverless-arkkitehtuuri, AWS, ohjelmistotestaus, versiohallinta, lambda-funktiot

Field of Study

Technology, Communication and Transport

Degree Programme

Tietotekniikan tutkinto-ohjelma

Author(s)

Daniel Mäkinen

Title of Thesis

Ohjelmistokehitys serverless-arkkitehtuurissa

Date 24.03.2024 | Pages/Appendicies 26 / 0

Client Organisation/Partners)
Vincit Oyj - Tampereen Särkänniemi Oy

Abstract

The purpose of the thesis was to document the software development process in a system based on serverless architecture while I was developing a new feature for it. The goal was to describe how development processes differ when working with this type of architecture compared to more traditional server-based architectures, i.e., which processes are streamlined and what different challenges serverless architecture presents. The project in question relied heavily on services hosted in Amazon Web Services, and therefore this paper also focuses a lot on their offerings.

First, we go through how the software architecture is defined and what serverless architecture means in this context. Then we describe the structure of the system being developed, followed by an explanation of the workflow stages encountered during the development of the feature. Finally, I will summarize my thoughts on the general themes that emerged in the development of serverless architecture and the insights gained about the architecture after the completion of the function.

Keywords

cloudservices, software development, serverless-architecture, AWS, software testing, version control, lambda-functions

SISÄLTÖ

1 Johdanto 4

2 Teoriaa Serverless-arkkitehtuurista 5

2.1 Mikä on ohjelmistoarkkitehtuuri? 5

2.2 Serverless-arkkitehtuurin ominaispiirteitä 5

2.3 Serverless- vs perinteinen arkkitehtuuri 7

3 Kuvausta työstettävästä järjestelmästä 11

3.1 Yleiskuvaa järjestelmästä 11

3.2 Backend 12

3.3 Frontend 13

4 Toteutuksen kuvaus 15

4.1 Yleiskuvausta toteutettavasta ominaisuudesta 15

4.2 Käyttöliittymä 16

4.3 Tietokanta muutokset 17

4.4 Cognito tunnistautuminen 17

4.5 Rajapintojen luonti 18

4.6 Lambdojen luonti 19

4.7 Sähköpostipalveluiden integrointi 21

4.8 Organisaatiopino Cloudfrontieriin 22

4.9 Valmis toiminto. 23

5 Loppumietelmiä 24

5.1 Ajatuksia serverless-arkkitehtuurista 24

5.2 Kokemuksia ja kiitoksia 25

A Lähdeluettelo 26

1 JOHDANTO

Kun minulle tuli ajankohtaiseksi suorittaa opinnäytetyöni, työskentelin konsulttitalo Vincit Oy:llä projektissa, jossa asiakkaana oli Särkänniemen huvipuisto. Vincit auttoi Särkänniemeä kehittämään digitaalista lippujärjestelmää sekä muita tietoteknillisiä asioita. Lippujärjestelmässä oli mielenkiintoista se, että järjestelmä oli toteutettu serverless-arkkitehtuurilla, johon en ollut aikaisemmin törmännyt muualla kuin terminä verkossa selaillessani.

Särkänniemi antoi minulle tehtäväksi luoda uuden toiminnon lippujärjestelmäänsä. Heidän yritysasiakkaillaan oli tarve jakaa lippuja digitaalisesti työtekijöilleen, ja minun tuli valmistaa käyttöliittymä sekä järjestelmän osat, jotka mahdollistaisivat tämän. Olin aiemmin pääasiassa muokannut olemassa olevia järjestelmän osia, ja nyt minulle tarjoutui mahdollisuus toteuttaa kokonaan uusia. Tässä edellytti syvällisempää perehtymistä järjestelmän yksityiskohtiin, jotta pystyisin rakentamaan uutta infrastruktuuria, jonka varassa uusi palvelu voisi toimia.

Tämä tarjosi myös sopivan mahdollisuuden saattaa opintoni loppuun ja asiakkaan kanssa sovittiin, että toteutan opinnäytetyöni sovelluskehityksestä serverless-arkkitehtuurissa. Pyrin työssäni kuvailemaan, mikä serverless-arkkitehtuuri ylipäätään on, millaisia asioita tarvitsee ottaa huomioon arkkitehtuuria työstäessä sekä mitkä ovat sen vahvuudet ja heikkoudet. Työn tarkoitus on auttaa hahmottamaan, mihin järjestelmää kehitettäessä ja laajennettaessa tulisi erityisesti kiinnittää huomiota, jotta serverless-arkkitehtuurin mukaisesti skaalautuessa ei kohdata vaikeuksia.

Järjestelmä pohjautuu hyvin vahvasti Amazon Web Services (AWS) -palveluihin, joten tässä opinnäytetyössä tarkastelen asioita useasti AWS:sän näkökulmasta. Muut palveluntajoajat omistavat vastaavanlaisia palveluita, joten tässä läpi käydyt asiat pätevät myös laajemmalti pilvipalvelujen maailmaan. Lisäksi kirjoitelmassa käsitellään myös joitain yleisiä ohjelmistokehitystä tukevia asioita, jotka todennäköisesti helpottavat myös muiden arkkitehtuurien pohjalta rakennettujen järjestelmien kehittämistä ja ylläpitämistä.

2 TEORIAA SERVERLESS-ARKKITEHTUURISTA

2.1 Mikä on ohjelmistoarkkitehtuuri?

Ohjelmistoarkkitehtuurista on useita eri määritelmiä, mutta koin kirjan 'Software Architecture in Practice ((2nd edition), Bass, Clements, Kazman; AddisonWesley 2003)' määritelmän selkeimmäksi:

> "Tietokoneohjelman tai laskentajärjestelmän ohjelmistoarkkitehtuuri tarkoittaa järjestelmän rakennetta tai rakenteita, jotka koostuvat ohjelmistoosista, niiden ulkoisesti näkyvistä ominaisuuksista ja näiden osien välisistä suhteista."

Arkkitehtuuri siis käsittelee sitä, miten kokonaisuuden osat vuorovaikuttavat keskenään. Osien "ulkoisesti näkyvät ominaisuudet"ovat keskeisessä osassa ja näillä tarkoitetaan omaisuuksia, minkä perusteella osan kanssa voidaan vuorovaikuttaa. Tähän kuuluu esimerkiksi osan rajapinta, mitä kautta se tarjoaa informaatiota muille osille tietyssä määritellyssä muodossa. Vastakohtana tälle on osan sisäinen implementaatio, joka ei ole näkyvissä muille osille eikä siksi ole oleellista tässä kontekstissa.

Jokaisella järjestelmällä on jonkinlainen arkkitehtuuri, sillä jokaisen järjestelmän voi osoittaa koostuvan keskenään vuorovaikuttavista osista. Tämä pitää paikkansa riippumatta siitä, onko arkkitehtuurin muodostus ollut tarkoituksellista tai että ymmärtävätkö järjestelmän kehittäjät itse järjestelmänsä arkkitehtuuria. Yleensä kuitenkin arkkitehtuurin piiriin kuuluu suuria rakenteellisia päätöksiä, joita on hyvin vaikeaa tai kallista lähteä muuttamaan jälkeenpäin, joten käyttötarkoitukseen sopivaa arkkitehtuuria koitetaan usein hahmotella jo uuden ohjelmistoprojektin alkuvaiheissa. Arkkitehtuurin valinnassa otetaan huomioon tekijöitä kuten esimerkiksi ohjelmiston nopeusvaatimukset, luotettavuustarpeet, skaalautuvuusmahdollisuudet tai tietoturvakysymykset.

2.2 Serverless-arkkitehtuurin ominaispiirteitä

Serverless-arkkitehtuurissa infrastruktuurin asennus ja ylläpito ulkoistetaan pilvipalveluntarjoajille (Amazon Web Services Inc, n.d.). Tällöin ei tarvitse käyttää omia resursseja palvelinten ja verkotusten suunnitteluun, fyysiseen toteutukseen ja ylläpitoon, jolloin säästetään aikaa ja rahaa. Serverless-alustoilla voidaan vuokrata juuri sen verran

laskentatehoa kuin sillä hetkellä tarvitaan, ja lähes poikkeuksetta on tarjolla palveluita, jotka mahdollistavat laskentatehon määrän automaattisen skaalautumisen nykyisen käyttötarpeen mukaan. Tällöin maksetaan vain siitä laskentatehosta, mikä on kullakin hetkellä tarpeellista ja tämä tapahtuu pilvipalvelun tarjoajan toimesta, eli asiakkaan ei tarvitse huolehtia tästä alkuperäisten asetusten jälkeen. Pilvipalveluntarjoajat yleensä hallinnoivat CDN:ää, mikä tarkoittaa sitä, että palveluntarjoaja omistaa palvelimia useassa paikassa ympäri maailmaa ja voi näissä palvelimissa suorittaa muun muassa sivujen välimuistittamista sekä kuormantasausta. Palvelimien geograafisesti läheinen sijainti suhteessa käyttäjiin ja välimuistitus auttavat vähentämään sivujen käsittelyssä tapahtuvaa viivettä.

Infrastruktuurin luovuttamisella on myös haittapuolensa. Tallennetusta tiedosta ei olla enää täysin kontrollissa ja täytyy luottaa pilvipalveluntarjoajan kykyyn hoitaa tietoturva kuntoon. Voidaan myöskin jäädä täysin riippuvaiseksi kyseisestä palveluntarjoajasta, sillä sen vaihtaminen on usein kallis ja työläs prosessi, ja tällöin ollaan jumissa kaikkien niiden puutteiden kanssa, joita kyseisen yrityksen rakenteissa on.

Infrastruktuurin lisäksi voidaan myös kokonaan ulkoistaa tiettyjä palveluita. Monet pilvipalveluntarjoajat tarjoavat valmiita backend-ratkaisuja jotka integroituvat natiivisti valittuun pilvipalveluympäristöön. Nämä ulkoistetut palvelut tunnetaan nimellä 'Backend as a Service' (BaaS), ja yleisimpiin palveluihin kuuluvat esimerkiksi tietokannanhallinta-, tietovarasto- ja autentikointipalvelut (Cloudflare Inc, n.d.).

Jos halutaan toteuttaa palvelut itse, voidaan käyttää tilattomia ja tapahtumakäynnistettäviä funktioita. Tätä kutsutaan nimellä 'Functions-as-a-service' (FaaS). Kehittäjien ei tarvitse huolehtia palvelinlogiikasta, vaan he voivat pelkästään keskittyä kirjoittamaan funktioita, jotka käynnistyvät tietyn tapahtuman, kuten rajapintakutsun, seurauksena (IBM Corporation, n.d.). Nämä funktiot ovat tilattomia, eli joka kerta kun niitä kutsutaan, niin niistä käynnistyy uusi identtinen instanssi. FaaS-toteutukset hyötyvät serverless-arkkitehtuurin yleisistä eduista, kuten skaalautuvuudesta ja vain käyttöasteen mukaisista maksuista.

Vaikka serverless-toteutuksissa fyysisen infrastruktuurin hallinta on kokonaan ulkoistettu, eri osien tarvitsee silti kommunikoida keskenään. Esimerkiksi rajapinnan voidaan haluta käynnistävän FaaS-funktion, tai funktion tarvitsee kutsua tietokantaa. Näiden eri serverless-komponenttien relaatioita voidaan kuvata erilaisissa infrastruktuuritiedostoissa, joissa määritellään miten eri osat kommunikoivat keskenään. Tätä kutsutaan 'Infrastructure as Code' (IaC) -menetelmäksi ja eri palvelut voivat käyttää erilaisia standardeja infrastruktuurin kuvaamiseen. Infrastruktuurin kirjoittamisen koodina

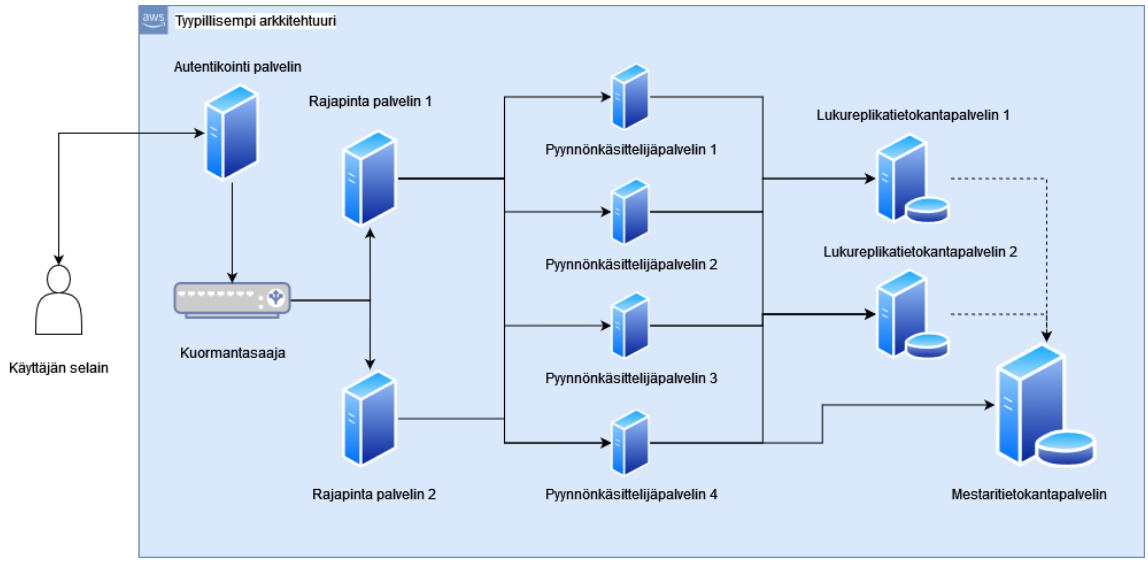
mahdollistaa johdonmukaisen ja toistettavan tavan ohjelmiston käyttöönotolle. Se luo myös keskitetyn paikan infrastruktuurin hallinnalle, joka voidaan sijoittaa esimerkiksi versionhallintaan, helpottaen muutosten seurantaa.

Serverless-arkkitehtuuri toimii parhaiten ratkaisuna ohjelmistoille, jotka toimivat tapahtumapohjaisesti, hyödyntävät lyhyen työmäärän toimenpiteitä, vaativat joustavaa skaalautuvuutta tai eivät tarvitse pysyvää tilaa (persistent state) funktiokutsujen välillä. Tapahtumapohjaisuus mahdollistaa tapahtumakäynnistettävien funktioiden käytön, mikä auttaa optimoimaan käyttöastetta ja sitä kautta vähentämään kuluja. Esimerkkeinä mahdollisesta sopivasta ohjelmistoista voisi olla verkkosovellus, jonka käyttöaste skaalautuu viikonpäivän mukaan ja jonka backendiä kutsutaan käyttäjän tapahtumien mukaisesti. Toinen mahdollinen kandidaatti voisi olla esineiden internet (IoT) sovellus, jonka toiminnot käynnistyvät erilaisten sensoritietojen perusteella. Vastaavasti sovellukset, jotka vaativat pitkään suoritettavia funktioita, kuten videoenkoodausta tai koneoppimista suorittavia funktioita, eivät sovellu yhtä hyvin serverless-arkkitehtuuriin (Cloudflare Inc, n.d.).

2.3 Serverless- vs perinteinen arkkitehtuuri

Tehdään katsaus esimerkkitilanteeseen, jossa meillä on kaksi erilaista backend-toteutusta. Toinen on toteutettu tyypillisemmällä arkkitehtuurilla ja se on voitu isännöidä joko paikallisesti tai pilvipalvelussa, ja toinen on toteutettu serverless-arkkitehtuurilla. Toteutuksessa kuvataan backendiä, joka autentikoinnin jälkeen käsittelee jonkin rajapintakutsun. Rajapintakutsun käsittelyssä tietoa haetaan tai kirjoitetaan tietokantaan, minkä jälkeen vastaus palautetaan käyttäjälle. Rajapintoja kutsutaan tiheään tahtiin ja niiden on silti oltava tavoitettavissa.

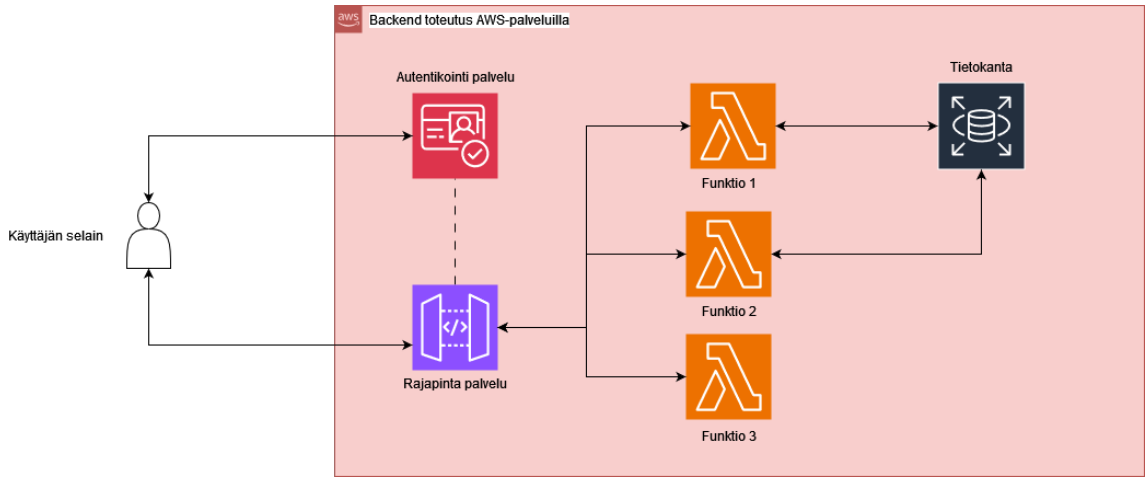
Alla on kuva tyypillisemmästä arkkitehtuurista, jossa infrastruktuuria hallitaan itse:



Kuva 1: Rajapintakutsun käsittely tyypillisessä arkkitehtuurissa

Kuvan palvelimet voidaan isännöidä joko fyysisesti organisaation tiloissa tai pilvipalveluissa. Tyypillisemmässä arkkitehtuurissa palvelimia hallitaan itse, ja niissä on huolehdittava ajettavan koodin ajoympäristön pystyttämisestä. Palvelinten on osattava ohjata saapuva tietoliikenne oikeisiin paikkoihin käsittelyä varten ja edelleen ohjata käsitelty liikenne eteenpäin. Mikäli tietoliikennettä on paljon, on tyypillisesti tarpeen käyttää kuoramantasaajaa. Sillä voidaan jakaa tulevaa liikennettä tasaisesti useammalle palvelimille, jotka ovat vapaina käsittelemään pyyntöä. Kuormantasaajan takana olevat palvelimet ovat yleensä kopioita toisistaan, ja esimerkiksi kaikki kuvan pyynnönkäsittelijäpalvelimet todennäköisesti sisältäisivät täydellisen kopion koko pyynnönkäsittelylogiikasta. Jos yksi palvelin kaatuu, muut voivat ottaa sen vastuut kokonaan. Tietokantapalvelinten kanssa skaalautuminen taas tapahtuu usein niin, että luodaan useita lukukopioita, jotka käsittelevät tietokannan lukupyyntöjä. Lisäksi on yksi pääpalvelin, jossa tehdään kaikki tietokantaan kirjoitettavat muutokset ja joka ylläpitää tietokannan ajantasaisia tietoja.

Seuraavaksi alla on kaavio mahdollisesta backend toteutuksesta Amazon Web Servicen (AWS) tarjoamilla serverless-työkaluilla:



Kuva 2: Rajapintakutsun käsittely serverless-arkkitehtuurissa

Tässä toteutuksessa perusrakenne on samankaltainen, mutta palvelimet on korvattu palveluilla. Pyynnönkäsittelijäpalvelimet on vaihdettu yksittäisiksi funktioiksi, joista jokainen käsittelee yhtä tiettyä kutsutyyppiä, sen sijaan että ne sisältäisivät aivan kaiken pyynnönkäsittelyyn liittyvän logiikan. Sen sijaan, että huolehditaan palvelinten välisestä tietoliikenteestä ja sen ohjautumisesta oikeisiin paikkoihin, eri palveluille kerrotaan, mihin toiseen palveluun halutaan yhdistää yhden palvelun ulostulo. Tämä on usein yksikertaisempaa, sillä palvelut on suunniteltu integroitumaan keskenään ja liitos saatetaan saada aikaan yhdellä rivillä koodia. Kuormantasaaja ei ole tarpeellinen, koska jokainen palvelu hoitaa oman skaalautumisensa tarpeen mukaan. Pilvipalveluihin isännöityä tyypillistä arkkitehtuuriakin voidaan skaalata automaattisesti, mutta tämä ei ole yhtä tehokas ratkaisu kun yksittäisen palvelun skaalaaminen. Esimerkiksi jos jokin tietty kutsutyyppi kohtaa valtavan ruuhkan, serverless-arkkitehtuurissa kyseisen kutsun käsittelijäfunktio skaalautuu yksinään, mikä on tyypillisesti nopeampi ja vähemmän resursseja vaativa operaatio kuin kokonaisen pyynnönkäsittelijäpalvelimen skaalautuminen. Skaalautuminen tapahtuu pellin alla automaattisesti, eikä ole tarvetta itse asentaa ja hallinnoida kuormantasaajia, vaikka tosin usein on mahdollista päästä hienosäätämään skaalautumista mikäli oletusasetukset eivät vastaa tarpeita.

Serverless-arkkitehtuurin toteutus on yleensä nopeampaa ja sen ylläpito vaatii vähemmän työtä. Palveluntarjoaja hoitaa infrastruktuurin yksityiskohdat ja suurimman osan sen ylläpidollisista asioista, kuten esimerkiksi ajoympäristöjen saavutettavuudesta ja ajantasaisuudesta. Toisaalta, yksittäisen tarjoajan palveluihin sitoutuminen tekee palveluntarjoajan vaihtamisesta hyvin haastavaa ja lukitsee toiminnallisuuden täysin pal-

veluntarjoajan tajonnan piiriin. Itsehallitut palvelimet yleensä pystytetään levykuvatiedostoista, jotka voidaan pystyttää toisen palveluntarjoajan palveluihin helposti. Lisäksi palvelimia itse hallinnoimalla ei ole rajoitettuna palveluntarjoajan asettamiin rajoitteisiin, vaan asennuksia ja asetuksia voidaan mukauttaa palvelimen käyttöjärjestelmän ja käytettävien teknologioiden puitteissa. Serverless-mallissa myös ohjelmistokehitysprosessi voi olla rajoittuneempi, koska todellisen lokaalin testiympäristön puuttuminen voi tuoda mukanaan haasteita, joita käsitellään lisää myöhemmin tässä kirjoitelmassa.

3 KUVAUSTA TYÖSTETTÄVÄSTÄ JÄRJESTELMÄSTÄ

3.1 Yleiskuvaa järjestelmästä

Työtehtävänäni oli rakentaa lisätoiminto Särkänniemen serverless-arkkitehtuuriin pohjautuvaan järjestelmään. Järjestelmä on laaja ja koostuu useista eri tarkoituksiin luoduista käyttöliittymistä, taustajärjestelmistä ja useista ulkoisista palveluista. Mainitsen tässä vain ne osat, jotka ovat oleellisia serverless-arkkitehtuurin taikka toteuttamani ominaisuuden kannalta. Keskeisimmät osat olivat lippujärjestelmä-backend, jota kutsun jatkossa vain nimellä backend ja jossa käsitellään lippu- ja tilausdataa, sekä Särkänniemen asiakkaille näkyvät julkiset verkkosivut, joihin lopulta lisäsin toiminnon käyttöliittymän. Backend isännöidään Amazon Web Services -palvelussa ja verkkosivut Vercelissä. Vercel on pilvipalveluntarjoaja, joka pilvipalveluiden tarjonnan lisäksi ylläpitää Next.js web-sovelluskehystä.

Eri osien lähdekoodit ovat versionhallinnassa, jossa niille suoritetaan jatkuvan integraation tarkistukset muutosten yhteydessä. Tarkistusten yleisimpänä tehtävänä on ylläpitää koodin laatua ja yhtenäisyyttä linterien avulla sekä varmistaa yksikkö- ja integraatiotestit ajamalla, ettei muutosten yhteydessä ole hajonnut mitään olemassa olevaa toimintoa. Järjestelmän osasta riippuen integraation yhteydessä saatetaan tarkistaa muita kyseiselle osalle oleellisia asioita, kuten esimerkiksi onko versionumeroita inkrementoitu tai onko jotain tiettyjä oleellisia kenttiä muistettu täyttää eri tietotyypeille. Lisäksi varmistetaan, että osat kääntyvät oikein ilman virheitä tai varoituksia.

Osalle järjestelmästä on määritelty omat infrastruktuuritiedostot, joiden perusteella ohjelman käyttöönotto tapahtuu kun tehdyt muutokset liitetään versionhallinnan päähaaraan. Näissä tiedostoissa määritellään, miten eri resurssit konfiguroituvat pilvipalveluissa. Integraatiotiedostot auttavat varmistamaan, että kriittisten osien käyttöönotto tapahtuu aina samalla tavalla. Mikäli muutosten yhteydessä ilmenee virheitä, versionhallinnan avulla voidaan helposti palata takaisin aiemmin toimivaksi todettuun tilaan.

Järjestelmästä ylläpidetään pääasiassa kahta eri ympäristöä, staging-ympäristöä asioiden testaamiseen ja kehitystyön tukemiseen, sekä production-ympäristöä, jota käytetään tuotannossa ja jossa liikkuu oikeat tuotteet sekä asiakasdatat. Eri ympäristöt toimivat eri domaineissa, ja jokaiselle uudelle ulkoiselle palvelulle on luotava erikseen molemmat ympäristöt, jotka ovat yhteydessä vastaaviin ympäristöihin järjestelmässä. Uusille sisäisille osille tarvitsee vain luoda uudet ympäristömuuttuja eri ympäristöille.

3.2 Backend

Backend muodostaa projektin suurimman yhtenäisen kokonaisuuden, ja se on rakennettu useiden eri AWS-palveluiden varaan. Eri käyttöliittymien ja ulkoisten järjestelmien tiedonsiirto tapahtuu pääasiassa tämän järjestelmän kautta, minkä vuoksi lähes aina uuden toiminnon kehittämisen yhteydessä on tarpeen tehdä muutoksia myös backend-puolella. Työstämääni toimintoon liittyen seuraavat kuusi AWS-palvelua ovat olleet keskeisiä:

CloudFormationille syötetään infrastruktuuritiedostoja, joiden avulla järjestelmän käyttöönotto (deployment) tapahtuu. Resurssit voidaan järjestää omiin "pinoihinsa", jotka kattavat aina yhteen kokonaisuuteen liittyvät resurssit. Kun jotain osaa järjestelmästä muutetaan, tarvitsee vain päivittää se pino mihin osa liittyy. Tämä nopeuttaa päivitysten käyttöönottoa ja ongelmatilanteissa vain tiettyjen osien palauttamista aikaisempaan versioon. Tässä projektissa CloudFormationille syötetyt infrastruktuuritiedostot on luotu Amazonin tarjomalla aws-cdk:lla (Cloud development kit), joka on saatavilla JavaScriptille Node-pakettina. Amazon tarjoaa CDK:n myös monille muille ohjelmointikielille.

AWS Lambda tajoaa tapahtumakäynnistettäviä ja tilattomia funktioita, joiden varaan backendin toiminta pääasiassa perustuu. Lambda-funktioiden luonti tapahtuu Amazonin tarjoamilla SDK:illa (Software Development Kit), joita on myös saatavilla useille eri ohjelmointikielille. Funktioille voidaan valita eri lähteitä, mitkä käynnistävät funktion, kuten esimerkiksi API-kutsu tai ajastin. Lambda-funktiot ovat tilattomia, mutta ne pystyvät kommunikoimaan ja hakemaan pysyvää dataa muista AWS-palveluista, kuten tietokannoista.

API Gateway tarjoaa järjestelmälle rajapinnat, joita kutsutaan eri käyttöliittymistä ja palveluista. CloudFormation luo perustan rajapinta-infrastruktuurille, ja käyttöönoton yhteydessä versionhallinnassa sijaitsevasta Swagger-tiedostosta haetaan määritellyt rajapintojen päätepisteiden konfiguraatiot. Swagger-tiedostossa on YAML-formaatissa yksityiskohtaiset määritykset, kuten esimerkiksi vastaanotettavat parametrit, mahdolliset vastaukset kutsuihin, autentikointi vaatimukset sekä tiedot siitä, minkä Lambda-funktion tai muun palvelun rajapintakutsun on tarkoitus käynnistää.

SQS-viestijono mahdollistaa ohjelman eri osien irtikytkennän (decoupling) toisistaan, mikä helpottaa osien muokkausta ja skaalausta erillisinä yksikköinä. Esimerkiksi erilaiset rajapintakutsut voivat käynnistää eri lambda-funktioita, jotka lähettävät dataa samaan SQS-jonoon. Jonon toisessa päässä oleva Lambda-funktio käsittelee sitten tä-

tä dataa saapumisjärjestyksessään. Tämä mahdollistaa lähettäjien ja vastaanottajien skaalauksen eri suhteissa käyttöasteen mukaan.

Amazon RDS tarjoaa hallittuja tietokantoja. RDS mahdollistaa tietokantojen skaalautuvuuden käyttöasteen mukaan sekä automaattiset varmuuskopiot. Nämä tietokannat ovat SQL-kantoja, ja niiden rakenne määritellään koodin puolella knex-kirjaston avulla luoduilla migraatiotiedostoilla. Nämä tiedostot kertovat, kuinka tietokannan rakenne muuttuu ajan saatossa. Esimerkiksi, kun tarvitaan uusi kenttä johonkin tietokantatauluun, luodaan uusi migraatiotiedosto, jossa määritellään, kuinka kenttä lisätään ja mahdollisesti poistetaan käyttöönoton yhteydessä.

AWS Cogniton avulla voidaan luoda autorisointi- ja autentikointiprosessit. Se tarjoaa käyttäjien luonti- ja kirjautumispalvelut sekä tukee ulkoisten palveluntarjoajien, kuten Googlen tai Facebookin, kirjautumisia. Cognito huolehtii käyttäjätietojen automaattisesta tallentamisesta oleellisimpien tietosuojastandardien mukaisesti. AWS-palvelut toimivat yleensä keskenään sulavasti, ja Cogniton autentikointi on helposti yhdistettävissä API Gatewayn kautta luotuihin rajapintoihin.

Järjestelmä hyödyntää AWS-palveluiden lisäksi useita muita ulkoisia palveluita. On tärkeää pystyä kommunikoimaan asiakkaiden kanssa sähköpostin kautta, ja esimerkiksi ostetut liput lähetetään sähköpostitse heti oston yhteydessä. Sähköpostien hallintaan on käytetty SendGrid-palvelua, joka tarjoaa myös JavaScript-kirjastoja palveluilleen. Sähköposti templaatteja voidaan hallita SendGridin verkkosivuilta ja JavaScript-kirjaston avulla voidaan toteuttaa automatisoituja sähköpostilähetyksiä lambda-funktioiden aktivoituessa.

3.3 Frontend

Käyttöliittymät on isännöity Vercelissä ja ne pohjautuvat Next.js -kehykseen. Vercel tarjoaa serverless-palveluille tyypillisiä ominaisuuksia, kuten palvelimien ylläpidon ja hallinnan, CDN-tuen suorituskyvyn parantamiseksi sekä automaattisen skaalauksen käyttöasteen mukaan. Versionhallinnassa ylläpidetään React-koodia, ja Vercel on asetettu päivittämään automaattiseti isännöimäänsä koodia versionhallinnassa tapahtuvien muutosten yhteydessä. Lisäksi Vercel tarjoaa muun muassa mahdollisuuden kääntää eri versionhallinnan haaroista (branch) testiversioita, joiden avulla voidaan tarkastaa koodin toimivuus Vercel-ympäristössä ennen lopullista koodin puskua. Vercel tarjoaa myös analytiikka- ja monitorointitoimintoja, joiden avulla voidaan jatkuvasti seurata käyttöliittymien tilaa ja toimivuutta.

Osa käyttöliittymistä hyödyntää ulkoista Contentful-CMS:ää (Content Management System), jonka avulla sivujen sisältöä voidaan hallita helposti ilman, että koodia itsessään tarvitsee muokata. Tämän asiosta Särkänniemen sisältövastaavat voivat muuttaa sisältöä itsenäisesti ilman, että kehitystiimin tarvitsee puuttua tilanteeseen.

4 TOTEUTUKSEN KUVAUS

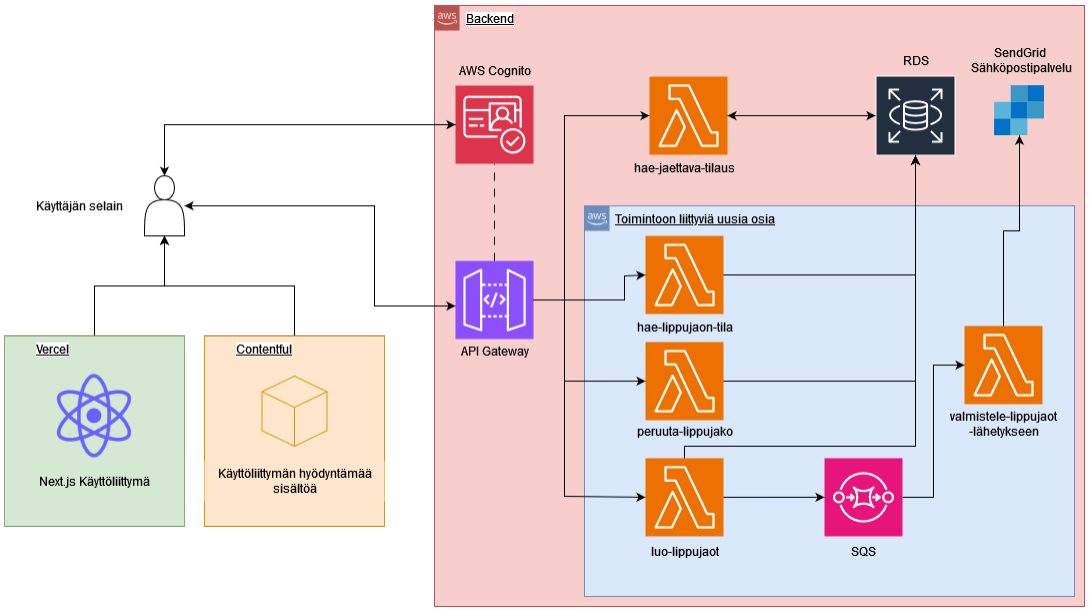
4.1 Yleiskuvausta toteutettavasta ominaisuudesta

Tehtävänäni oli toteuttaa lippujen jakotoiminto, jota Särkänniemen yritysasiakkaat voisivat käyttää huvipuistolippujen digitaaliseen jakamiseen työntekijöilleen. Tarkoituksena oli luoda käyttöliittymä, johon voidaan ladata Excel-tiedosto, joka sisältää vastaanottajien sähköpostiosoitteet sekä heille tarkoitettujen lippujen määrän. Näiden tietojen perusteella lippuja jaellaan määriteltyjen vastaanottajien sähköposteihin. Lisäksi käyttöliittymän tulee mahdollistaa Excelistä haetun tiedon muokkaaminen ennen lopullista lähetystä sekä yksittäisten vastaanottajien lisääminen. Toiminnon on oltava kirjautumisen takana, koska järjestelmän kautta liikkuu merkittäviä summia rahaa lippujen muodossa. Lisäksi on oltava mahdollisuus mitätöidä lähetetyt liput. Jos liput lähetetään vahingossa väärään osoitteeseen, tulee olla mekanismi, jonka avulla lähetetyt liput voidaan mitätöidä ja lähettää uudelleen oikeaan osoitteeseen.

Tehtävä syntyi, kun Särkänniemi kertoi useiden yritysasiakkaiden ilmaisseen tarpeen päästä jakamaan lippuja helposti työntekijölleen ja kuinka yritysasiakkaat ovat käyttäneet Exceliä jaettavien lippujen kirjaamiseen. Projektimme pääsuunnittelija hioi halutut ominaisuudet yhdessä Särkänniemen kanssa ja suunnitteli alustavat kaaviot ja kuvat mahdollisesta käyttöliittymästä Figma-sovelluksessa. Kun tarpeet oli selvitetty ja käyttöliittymästä suunnitelmat olemassa, työ siirtyi minulle toteutettavaksi.

Toteutettu työ ei sinänsä ole itsessään oleellinen tämän kirjoitelman aiheen kannalta, ja mainitsen toiminnosta vain asioita, jotka koen lisäävän tekstin ymmärrettävyyttä. Muuten pyrin kertomaan hyvin yleisellä tasolla, minkä luonteisia prosessin eri vaiheet olivat ja miten serverless-arkkitehtuuri mahdollisesti muuttaa työnkulkua verrattuna perinteiseen serveripohjaiseen arkkitehtuuriin.

Alla on kuva, joka kuvastaa osaa relevanteista järjestelmän osista. Sinisellä pohjalla olen kuvannut joitain uusia osia, joita toiminnon toteuttaminen vaatisi.



Kuva 3: Järjestelmään liitettäviä osia

Backendiin oli siis lisättävä funktioita, jotka käsittelivät lippujakoihin liittyviä tapahtumia ja muutoksia. Jo olemassa olevia backendin osia oli myös muokattava tukemaan näitä operaatioita. Käyttöliittymä piti luoda uudeksi osaksi olemassa olevaa Vercelissä isännöityä Next.js-projektia.

4.2 Käyttöliittymä

Itse käyttöliittymän luominen tapahtui projektin loppuvaiheessa, mutta kaikki backend-muutokset kuitenkin suunniteltiin ottaen huomioon, miten käyttöliittymä myöhemmin toteutettaisiin, ja siksi mainitsen sen nyt tässä ensimmäisenä. Käyttöliittymän luomisprosessi erosi vähiten arkkitehtuurisista syistä, ja sen toteuttaminen oli perinteistä frontend-ohjelmointia React- ja Next.js-rakenteisiin.

Käyttöliittymästä kutsutaan API Gatewayssä määriteltyjä rajapintoja, jotka palauttavat tietoa ja suorittavat tarvittavia operaatioita. Selain tarkistaa aluksi kirjautumispolettien (login tokens) läsnäolon, ja mikäli näitä ei löydy, käyttäjä ohjataan kirjautumaan Cognito-palvelun tarjoamalle kirjautumissivulle.

Vercel-isännöinti mahdollisti keskeneräisen käyttöliittymän nopean muuntamisen julkiseksi verkkoversioksi. Tämä mahdollisti asiakkaalle suoran linkin tarjoamisen kyseiseen

versioon, mikä teki palautteen keräämisestä tehokkaampaa. Pystyin varmistumaan siitä, että asiat toimivat oikein ympäristössä, joka muistuttaa hyvin paljon lopullista tuotantoympäristöä.

4.3 Tietokanta muutokset

Toiminto vaati uusien tietokantataulujen luomista sekä muutoksia olemassa oleviin tauluihin. Projektissa tietokantahallintaa toteutetaan pääosin Knex-kirjaston avulla, joka tarjoaa työkalut migraatiotiedostojen luomiseen ja ajamiseen. Loin muutoksia varten tarvittavat migraatiotiedostot ja ajoin skriptin, jonka tarkoitus on avustaa tietokantamuutosten tekemistä paikallisessa ympäristössä. Aikaisemmin olin luonut skriptillä paikallisen Docker-kontin, joka ylläpitää rakenteellista kopiota tuotannon tietokannasta. Tätä kantaa käytetään migraatioiden aiheuttamien muutosten tarkistamiseen ja tietokantaa vastaavien TypeScript-tyypitysten päivittämiseen Schemats-kirjaston avulla. Nämä TypeScript-tyypitykset ovat hyödyllisiä lambda-funktioiden kehittämisessä, koska ne auttavat varmistamaan tietokannasta palautuvan tiedon tietotyypit.

Tietokantamigraatioille on luotu erillinen skripti integraatioputkeen, joka käynnistyy, kun versionhallintaan lähetetään uusi versio. Tämä skripti luo väliaikaisen kevyen Linux-järjestelmän, joka asentaa itselleen Noden sekä Knex-kirjaston ja ajaa tämän kirjaston avulla uuden version migraatiotiedostot oikeaan RDS-tietokantaan. Näin ollen uuden version lähettämisessä täytyy varmistua vain siitä, että migraatiotiedostot ovat olemassa ja toimivat oikein. RDS-tietokannan päivitys tapahtuu sen jälkeen automaattisesti.

4.4 Cognito tunnistautuminen

Työstämäni toiminto oli järjestelmän ensimmäinen erityisesti Särkänniemen yritysasiakkaille suunnattu ominaisuus, joten projektissa ei ollut valmiita rakenteita yritysasiakaskohtaisiin toimintoihin. Yritysasiakkaille täytyi luoda omat käyttäjäinfrastruktuurit kirjautumista varten, ja tämä onnistui hyödyntämällä AWS:n Cognito-palvelua.

Cognitoa määritellessä minun tuli päättää muun muassa seuraavia asioita: voivatko käyttäjät itse luoda tunnuksia, kirjaudutaanko sähköpostilla vai erillisellä käyttäjänimellä, mitä tietoja käyttäjiltä kerätään ja mitkä niistä ovat pakollisia, sekä kuinka pitkään kirjautumispoletit ovat voimassa. Palvelussa pystyy myös luomaan templaatit erilaisille kirjautumisiin liittyville sähköpostiviesteille, kuten liittymiskutsuille ja salasananvaihto-

viesteille.

Lisäksi minun piti määrittää, miten käyttäjät kirjautuvat palveluun. On mahdollista luoda ja isännöidä omat kirjautumis-sivut ja tähän Cognito tarjoaa työkaluja, muun muassa JavaScript-kirjaston muodossa. Toinen vaihtoehto on käyttää valmista AWS:n tarjoamaa kirjautumissivua, johon sitten voidaan linkittää työstettävästä käyttöliittymästä. Tämä kirjautumissivu mahdollistaa rajoitetun muokkauksen, kuten ulkoasun muokkauksen CSS-tiedoston avulla ja yrityslogojen lisäämisen. Onnistuneen kirjautumisen jälkeen kirjautumispoletit toimitetaan selaimelle, joka ohjataan siihen sivulle, joka on määritelty query-parametrina kirjautumissivulle. Nämä poletit tallennetaan väliaikaisesti selaimen muistiin ja niitä käytetään käyttäjän kirjautumistietojen ylläpitämiseen session aikana.

Cogniton avulla voidaan ulkoistaa suurin osa autentikointilogiikasta ja käyttäjätietojen hallinnasta suoraan Amazonille. Ei tarvitse itse talletella käyttäjätietoja omissa tietokannoissa, ja voidaan hyödyntää Cogniton integraatio-ominaisuuksia muissa AWS:n palveluissa. Esimerkiksi API Gatewaylle voidaan muutamalla koodirivillä Swagger-tiedostoon määritellä, että rajapinnat vaativat tästä Cognito käyttäjäjoukosta saadut kirjautumistunnukset HTTP-pyyntöjen otsikoissa. AWS huolehtii tämän jälkeen kirjautumistunnusten validoinnista ja palauttaa virheellisten tunnusten kohdalla selkeitä virheilmoituksia.

Tämän valmiiksi tarjotun palvelun käyttöönotto ja integrointi säästivät merkittävästi aikaa verrattuna oman kirjautumispalvelun kehittämiseen. Toisaalta tässä menetelmässä menetetään jonkin verran kontrollia, eikä voida vapaasti toteuttaa kaikkia toimintoja, joita saatetaan pitää hyödyllisinä. Cogniton palvelut olivat kuitenkin sen verran kattavat, että en missään vaiheessa kokenut tarvetta monimutkaisemmalle toteutukselle.

4.5 Rajapintojen luonti

Tässä projektissa API Gatewayn päätepisteiden määrittely tapahtuu Swagger-tiedostossa, joka on YAML-pohjainen määrittelytiedosto. API Gateway tukee natiivisti Swagger-tiedostoja ja osaa luoda näistä lopulliset rajapinnat automaattisesti. Amazon on myös lisännyt omia Swagger-parametreja ja -metodeita, joita tulee hyödyntää määrittelyjä tehdessä. Tiedostossa määritellään muun muassa mitä autentikointimetodeja rajapinnat vaativat, mitä parametrejä rajapinta voi vastaanottaa, sekä minkä lambdan tai muun palvelun rajapinta käynnistää pyynnön käsittelyä varten. Mikäli pyyntö ei vas-

taa määriteltyjä parametreja, API Gateway luo automaattisesti oikean virhekoodin sekä kuvauksen virheestä, joka ilmoitetaan pyynnön lähettäjälle. Esimerkiksi jos pyynnön runko (body) ei vastaa määriteltyä runkoa tai mikäli jokin oleellinen HTTP-pyynnän otsikko (header) puuttuu.

Mikäli pyyntö noudattaa asetettuja sääntöjä, se ohjataan määritellylle lambda-funktiolle, joka käynnistyy automaattisesti käsittelemään sitä. Kun funktio on käsitellyt pyynnön, se palauttaa vastauksen API Gatewaylle, joka välittää sen takaisin alkuperäiselle pyynnön lähettäjälle. Tämä eroaa serverikeskeisestä järjestelmästä siten, että pyyntöä ei ohjata millekkään tietylle palvelimelle tai kuormantasaajalle, joka sitten omien sisäisten rakenteiden perusteella käsittelisi pyynnön. Tässä tapauksessa kutsuja käsittelevää logiikkaa ei tarvitse isännöidä tai ylläpitää missään itse hallitussa järjestelmässä, vaan lambda-funktioiden luominen riittää ja AWS huolehtii ajoympäristöjen hallinnasta taustalla.

Swagger-cli sisältää työkaluja Swagger-tiedostojen oikeellisuuden tarkistamiseen. Projektissamme nämä tarkistukset ovat liitetty skriptiin, joka ajetaan automaattisesti versionhallintaa päivittäessä, jotta vältytään turhilta virheiltä. Toinen skripti taas loi Swagger-tiedostosta automaattisesti TypeScript tyypitykset, joita hyödynnetään lambdojen ja käyttöliittymän luonnissa.

Koko rajapintalogiikkaa voidaan ylläpitää näissä Swagger-tiedostoissa, ja missään vaiheessa ei ole tarvetta itse luoda minkäänlaista koodi-logiikkaa rajapinnoille. Tämä tekee rajapintojen hallinnasta helppoa, mutta toisaalta jäädään jonkin verran Swaggerin ja API Gatewayn rajoitteiden armoille. Tässäkin tapauksessa onneksi palveluiden tarjonta oli niin laajaa että tästä ei koitunut ongelmaa.

4.6 Lambdojen luonti

Ylivoimaisesti suurin osa järjestelmään luoduista lambda-funktioista käsittelee rajapintakutsuja. Swagger-tiedoston rajapinnoista luodaan TypeScript-tyypitykset dtsgen-työkalun avulla, ja näitä voidaan käyttää lambda-funktioiden pyyntöjen ja vastausten tyypittämiseen. Rajapintakutsua käsittelevä lambda saa argumenttinaan objektin, jossa on tietoa pyynnöstä, ja tästä objektista voidaan hakea tyypitetyt parametrit, joita käytetään pyynnön käsittelyssä.

Otetaan yksinkertainen esimerkkitapaus rajapinnasta, josta voidaan hakea tietyn lippujaon tila. Rajapintapyynnössä tulee parametrina sen lippujaon tunnus (id), jonka tila

halutaan hakea. Tämän tunnuksen avulla lambda-funktiossa suoritetaan tietokanta-haku, jossa haetaan kyseisen lippujaon tila. Tiedot muunnetaan Swaggerissa määriteltyyn vastausmuotoon, ja ne palautetaan API Gatewaylle ja siitä edelleen alkuperäisen pyynnön lähettäjälle. Alla oleva kuva näyttää esimerkin mahdollisesta lambda-toteutuksesta:

```
const fetchTicketShareStatus = async (event: apiEvent) => {
    const db = databaseConnection();

    const id = event.body.id;

    const [ result ] = await db("ticket_share").select("id", "status").where("id", id);

    const isUsable = ["registered", "active"].includes(result.status);

    return {
        id: result.id,
        isUsable,
    };
};

export default fetchTicketShareStatus;
```

Kuva 4: Yksinkertainen esimerkki lambda-funktiosta

Lambdat siis voivat olla yksinkertaisia JavaScript-funktioita, jotka saavat argumentteina tietoja käynnistävästä palvelusta ja joiden tulee palauttaa dataa tietyssä muodossa kutsuvalle palvelulle tai ohjata data eteenpäin seuraavalle palvelulle. Käytännössä kuitenkin lambdat ovat usein yksinkertaista esimerkkiä monimutkaisempia. Esimerkiksi tämä funktio voitaisiin laajentaa palauttamaan paljon enemmän tietoa lippujaosta.

Yksi haaste, joka eroaa perinteisestä backend-toteutuksesta, on löytää ratkaisu, jolla varmistetaan lambdan oikeellinen toimivuus lokaalin kehityksen aikana. Lopullisen toimivuuden näkee vasta sitten, kun kyseinen lambda on lähetetty pilveen osaksi aktiivista järjestelmää. Tyypillisemmässä arkkitehtuurissa pystytään usein pyörittämään lokaalia versiota koko backendistä, jossa voidaan testata asioiden toimivuus kehitysvaiheen aikana. Meidän projektissamme tarvitsi luottaa jest-testeihin. Jokaiselle lambdalle luodaan testit, joissa voidaan hyödyntää dtsgen-työkalulla luotuja tyypityksiä rajapinnasta. Tällöin on aina tiedossa, missä muodossa palautettava data halutaan, ja tätä tietoa hyödynnetään testien päämääränä. Testissä siis lambdalle syötetään erilaisia parametreja ja varmistetaan että saadaan lambdasta ulos oikeaa dataa, joka on Swaggerissa määritellyssä muodossa. Suoria kutsuja ei siis voida lähettää testausvaiheessa, mutta hyvin tehdyillä testeillä voidaan varmistaa suurin osa toimivuudesta.

Työstämääni toimintoa varten minun piti luoda useampia rajapinnoista kutsuttavia lambdoja, jotka kommunikoivat tietokannan kanssa. Lisäksi tarvitsin muutaman lambdan jotka käsittelivät sähköpostien lähettämisen ja sähköpostien tilojen seurannan, ja näitä käsitellään seuraavassa kappaleessa.

4.7 Sähköpostipalveluiden integrointi

Asiakkaille lähetettävät sähköpostit menivät Twilion omistaman SendGrid-palvelun kautta. Tavoitteena oli lähettää SendGridin avulla jaetut liput vastaanottajien sähköpostiosoitteisiin.

Sähköposteja varten luodaan ensiksi templaatit SendGridin verkkosivuilla tarjotuilla työkaluilla. Näihin templateihin voidaan määritellä muuttujia, jotka syötetään myöhemmin, kun sähköposteja lähetetään SendGridin JavaScript-kirjaston avulla. Muuttujat voivat sisältää esimerkiksi vastaanottajan nimen tai muita yksilöllisiä tietoja, jotka räätälöidään kullekin vastaanottajalle. Templaatteihin saa myös simppeleitä ehtolauseita, joiden perusteella voidaan esimerkiksi näyttää jokin erikoiskenttä.

Kun käyttäjä on tarkistanut lippujako-toiminnon käyttöliittymässä, että asiat näyttävät oikeilta ja on valmis lähettämään liput, kutsutaan rajapintaa. Tämän rajapinnan käynnistämä lambda-funktio luo lippujaot ja tallentaa lippujen tilan tietokantaan. Samalla tämä funktio välittää tietoa lähetettävistä lipuista SQS-jonoon. Toinen lambda lukee jonosta syötetietoa ja lähettää liput SendGridin tarjoaman kirjaston avulla. Operaatiossa voidaan lähettää kerralla useita tuhansia lippuja, jotka jaetaan pienempiin eriin ja syötetään järjestelmällisesti jonoon. Tämä mahdollistaa useiden lambda-funktioiden samanaikaisen käytön SQS-jonon lukemiseen ja varmistaa, että yhden lambdan virhetilanteessa vain osa lipuista siirtyy virhetilaan. Jono tallentaa myös epäonnistuneet liput, jotka voidaan myöhemmin yrittää lähettää uudelleen.

SendGrid tarjoaa myös palvelun lähetettyjen sähköpostien tilanseurannalle. Päätin hyödyntää tätä, jotta käyttäjä saa varmuuden siitä onko liput menneet perille. Sähköpostien virheellinen käsittely voi johtua esimerkiksi väärästä tai olemattomasta sähköpostiosoitteesta tai tietoliikenneongelmista, jotka estävät sähköpostin perillepääsyn. SendGrid on määritetty lähettämään HTTP-pyyntö aina, kun lähetetyssä postissa ilmenee jokin virhe.

Vastaanottaaksemme näitä ilmoituksia, minun oli rakennettava vastaanottava rajapinta. Swaggerissa määriteltiin mitä kenttiä SendGridin lähettämässä pyynnössä on ja

että turva-avaimet sisältävät otsikot ovat läsnä. Nämä otsikkotiedot välitetään lambdalle ja lambda-prosessin alussa tarkistetaan, että turvaotsikoiden tarkistussummat ovat oikein. Vasta tämän jälkeen sähköpostin statustieto tallennetaan tietokantaan. Näin varmistutaan, että kyseistä rajapintaa voi käyttää vain SendGridin palvelu.

4.8 Organisaatiopino Cloudfrontieriin

Kuten aikasemmin mainittiin, tämä oli ensimmäinen pääasiassa yrityskäyttöön luotu järjestelmän toiminto. Oli tarve koota kaikki toimintoon ja yritysinfrastruktuuriin liittyvät palaset yhdeksi CloudFrontier-pinoksi sujuvampaa ylläpitoa varten.

Projekti sisälsi valmiita rakenteita sille, miten infrastruktuuria hallitaan. Hyödynsin projektilaisten aws-cdk:lla luomaa pino-luokkaa, joka sisältää paljon erilaisia valmiita metodeja AWS-resurssien helppoon alustukseen. Luokka mahdollistaa esimerkiksi lambdojen liittämisen pinoon sekä ApiGateway rajapintojen luontiin tarvittavan Swagger-tiedoston haun ja alustuksen oikesta paikasta.

Valmistuttuaan pino mahdollisti sen, että jokaisen uuden lambdan luomisen, vanhan muokkaamisen tai Swagger-tiedoston päivittämisen yhteydessä, pinoon liittyvät resurssit päivittyvät automaattisesti myös AWS:sän päässä. Eli kun muokataan pinoon liittyviä resursseja ja pusketaan ne versionhallintaan, AWS:sälle lähtee päivitetty infrastruktuuri koodi, jonka avulla se muokkaa palveluita vastaamaa tiedoston määrittelyä. Näin ei tarvitse mennä muuttamaan asioita AWS:sään suoraan selaimella tai konsolilla, vaan muutokset versionhallinnassa automaattisesti muovaa infrastruktuuria AWS:sän palveluissa. Pino jaottelu myös varmistaa sen, että jos muutoksia tapahtuu vain yhden pinon osiin, pelkästää kyseinen pino päivitetään. Tällä säästetään aikaa ja rahaa, eli ei tarvitse maksaa pilvipalvelinten turhasta ajo-ajasta.

Kehitysvaiheessa aws-cdk:n TypeScript-tuki auttoi varmistamaan, että koodi oli syntaktisesti oikein. Sen sijaan koodin toiminnallisuutta ei taaskaan pystytty suoraan testaamaan paikallisesti. Kun näytti siltä että pinon kuuluisi toimia oikein, piti pino lähettää CloudFrontieriin staging-ympäristöön ajoon aws-cli:n avulla. Tämä konsolityökalu käynnisti pinonrakennusprosessin pilvessä ja informoi mahdollisista ongelmista. Kun pino saatiin onnistuneesti pilveen, voitiin testaan pinon rajapintoja ja autentikoitumisrakenteita staging-ympäristössä.

4.9 Valmis toiminto

Lopulta toiminto valmistui ja se siirtyi testattavaksi staging-ympäristöön. Siellä sitä stressitestattin luomalla todella suuria tilauksia, joista lähetettiin tuhansia lippujakoja. Varmistettiin, että lippujaot menivät suurissakin tilauksissa oikeisiin sähköpostiosotteisiin ja että ne sisälsivät oikeat tuotteet. Käyttöliittymästä pyydettiin myös viimeisiä käytettävyys arvioita ja näiden pohjalta tehtiin muutoksia, jotka tekivät toiminnon käyttämisestä yksinkertaisempaa.

Tällä hetkellä toiminto ei ole vielä ollut käytössä asiakasympäristössä. Toivotaan, että testaaminen oli tarpeeksi kattavaa ja että todellisessa käyttötilanteessa ei esiinny merkittäviä ongelmia.

5 LOPPUMIETELMIÄ

5.1 Ajatuksia serverless-arkkitehtuurista

Projektissamme serverless-arkkitehtuurin suurimmaksi haasteeksi muodostui toimivuuden varmistaminen ennen muutosten puskemista versionhallintaan. Tämä oli usein hankalaa ja johti tilanteisiin, joissa virheet huomattiin vasta todellisessa käyttöympäristössä. Vaikka huolellisesti tehdyt testit kaappasivat suurimman osan virheistä, ne eivät pystyneet simuloimaan kaikkia ajoympäristössä esiintyviä tilanteita. Välillä tuli myös vastaan tilanteita, joissa piti testata pieniä muutoksia ja sitä miten ohjelma reagoi niihin todellisessa ympäristössä. Sen sijaan että aina pienen muutoksen jälkeen tekisi vetopyynnön ja muuttaisi asian versionhallinnan kautta, tuli järkevämmäksi tavaksi mennä muuttamaan koodia tai parametrejä suoraan pilviympäristöön AWS:n tarjoamilla yksinkertaisilla editoreilla ja vasta oikeiden arvojen löytämisen jälkeen luoda muutos versionhallintaan. Kun näin sivuutetaan IaC-rakenteet ja tehdään suoraan muutoksia pilveen, syntyy riski siitä että pilviympäristöön jää jotain ongelmallista mitä on vaikeaa enään myöhemmin huomata, joten tätä ei mielellään tehtäisi mikäli tätä pystytään välttämään.

Toisaalta taas sitten serverless-arkkitehtuuri tarjosi helpon tavan lisätä ja poistaa toimintoja järjestelmästä. Infrastruktuurikoodi oli järjestetty niin, että uusia asioita oli todella nopeaa lisätä ja asioiden integroiminen keskenään oli mietitty pitkälle AWS:n toimesta. AWS on ollut jo aika pitkään olemassa ja heillä on ollut aikaa rakentaa kattava dokumentaatio, sekä lisätä monipuolisia toimintoja erillaisiin harvinaisempiinkin tarpeisiinkin, mikä helpotti kehitystyötä.

Loppujen lopuksi alkuperäiset projektin luojat ovat miettineet asiat hyvin ja järjestelmä sopii hyvin serverless-arkkitehtuuriin. Jatkossakin vain pitää aina huolta että palveluiden tarjonta on riittävää uudelle kehitykselle ennen uuden toiminnon luomisen aloittamista, ja jos ei ole, luo uuden järjestelmästä irrallisen palan joka sitten liitetään rajapintojen kautta. Tulevaisuuden kannalta voisi olla hyödyllistä harkita työkaluja pilviympäristön lokaaliin emulointiin. Vaikka kaiken emulointi on aikaa vievää ja epäkäytännöllistä, kriittisimpiä toimintoketjuja voitaisiin emuloida ja testata lokaalisti, mikä mahdollisesti auttaisi välttämään ongelmia ketjujen muokkaamisen yhteydessä.

5.2 Kokemuksia ja kiitoksia

Projektissa työskentely antoi minulle todella hyvää perspektiiviä siihen, miten serverless-arkkitehtuuri käytännön tasolla toimii. Tämän kokemuksen jälkeen pystyn paremmin arvioimaan, milloin uusia projekteja luodessa kannattaa harkita serverless-arkkitehtuuria helpottamaan ylläpito- ja kehitystyötä. Työstämäni järjestelmä oli myös ylipäätään mielenkiintoinen ja hyvin rakennettu kokonaisuus, mistä opin todella paljon yleisiä parempia ohjelmoinnin käytäntöjä.

Haluan vielä nopeasti antaa parit kiitokset. Kiitokset Vincitin Jussi Mustoselle ja koko hänen vetämälle kehitystiimille, jotka tarjosivat loistavan ympäristön oppia uutta ja tukea heiltä sai aina kun sitä pyysi. Kiitokset Särkänniemen Tuula Salmiselle, joka hoisi tuoteomistajan tehtiviä taitavasti. Kiitokset myös muulleekin Särkän porukalle, sillä kaikkien kanssa oli aina miellyttävä tehdä hommia. Kiitos Savonian Janne Koposelle rennosta ja ymmärtäväisestä tavasta ohjata opinnäytetyön etenemistä, vaikka välillä minun järjestelyt olivat vähän sekavat ja hämärillä aikatauluilla. Ja kiitos vielä Savonian opinto-ohjaajalle Sami Lahdelle, joka on yksi suurimmista syistä siihen, miksi opintoni etenivät tänne asti.

A LÄHDELUETTELO

Bass, L., Clements, P., & Kazman, R. (2003). Software Architecture in Practice (2nd ed.). Addison-Wesley.

Amazon Web Services, Inc. (ei päivämäärää). Serverless Architectures - Learn More [verkossa]. https://aws.amazon.com/lambda/serverless-architectures-learn-more/ [Viitattu maaliskuussa 2024].

Cloudflare, Inc. (ei päivämäärää). Backend as a Service (BaaS) [verkossa]. https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/ [Viitattu maaliskuussa 2024].

IBM Corporation (ei päivämäärää). Function as a Service (FaaS) [verkossa]. https://www.ibm.com/topics/faas [Viitattu maaliskuussa 2024].

Cloudflare, Inc. (ei päivämäärää). Why use serverless? [verkossa]. https://www.cloudflare.com/learning/serverless/why-use-serverless/ [Viitattu maaliskuussa 2024].