Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Johanna Tani

Serverless-arkkitehtuuri web-sovellus-kehityksessä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

18.6.2020

Tekijä Otsikko Sivumäärä Aika	Johanna Tani Serverless-arkkitehtuuri web-sovelluskehityksessä 37 sivua 18.6.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Juha Kämäri

Työssä perehdyttiin Serverless-arkkitehtuuriin ja sen toimivuuteen web-sovelluskehityksessä. Nimitys Serverless, eli palveliton, viittaa siihen, että pilvipalveluntarjoaja hoitaa liikenteen jakamisen tarjoajan ylläpitämille vapaille palvelimille. Pilvipalvelujen yleistymisen kautta Serverless-arkkitehtuuri on noussut suosioon helpottamalla sovellusten kehitystä, käyttöönottoa sekä ylläpitoa. Serverless mahdollistaa osittaisen vastuunsiirron pilvipalvelujen tarjoajille, jotka vastaavat näin ollen palvelinpuolen logiikasta, prosesseista ja allokaatiosta.

Työssä toteutettiin Serverless-arkkitehtuuria noudattava web-sovellus arkkitehtuurin mukaisesti. Sovelluksen kehitykseen käytettiin IBM Cloud -kehitysalustaa, joista tärkein käytetty palvelu oli IBM Cloud Functions. Testisovelluksen määrittelyssä asetettiin toiminnallisuuksille priorisointeja, jotta voitiin tutkia, joutuuko kehitettävistä toiminnallisuuksista karsimaan noudattamalla Serverless-arkkitehtuuria. Määrittelyjen avulla saatiin vertailukohde siihen, tuoko arkkitehtuurin noudattaminen etua sovelluskehitykseen vai luoko se turhia esteitä.

Testisovelluksen toteuttamisen jälkeen arvioitiin prioriteettimääritysten avulla, kuinka onnistuneesti haluttu sovellus kyettiin toteuttamaan. Asetetuista vaatimuksista puolet saatiin toteutettua kokonaan ja puolet osittain. Prioriteettijärjestyksen mukaan kolme onnistunutta vaatimusta olivat tärkeimmät ja kolme osittain onnistunutta järjestyksen lopussa. Laskemalla vaatimuksien onnistumisprosentit ja ottamalla huomioon priorisoinnin, saatiin 86-prosenttisesti onnistunut web-sovellus.

Lopuksi tuloksia hyödyntämällä pohdittiin mahdollisia Serverless-arkkitehtuuriin sopivia käyttökohteita. Samalla pohdittiin tuloksista huomattuja hyviä ja huonoja puolia arkkitehtuuriin sekä kehitysalustaan liittyen.

Avainsanat	Serverless, Serverless-arkkitehtuuri, IBM Cloud

Author Title Number of Pages Date	Johanna Tani Serverless Architecture in Web Application Development 37 pages 18 June 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Development
Instructors	Juha Kämäri, Senior Lecturer

The thesis takes a closer look at Serverless architecture and its functionality in web application development. The name Serverless refers to the cloud provider taking responsibility of providing the capacity of servers. With the use of cloud services, Serverless architecture has become more popular by facilitating application development, deployment and upkeep. Serverless enables partial move of responsibility to cloud service providers who then oversee the backend logic, processes and allocation of server capacity.

The thesis covers the implementation of a test web application that uses Serverless architecture. IBM Cloud development platform was used in the development. The test web application had requirements that were prioritized so those could be used to evaluate the development of the application. The successful development of the needed requirements meant that using the architecture was beneficial.

The test web application was evaluated using the prioritized requirements. Half of the requirements were successfully implemented, and the rest were partially implemented. The requirements with the highest prioritization were successfully implemented, and requirements with lower prioritization were the ones partially implemented. The test application was 86 percent successfully implemented.

The end results were used to consider possible use cases for Serverless architecture. Benefits and disadvantages regarding the architecture and development platform were also examined based on the end results.

Keywords	Serverless, Serverless architecture, IBM Cloud

Sisällys

Lyhenteet

1 Johdanto 1

2 Serverless-arkkitehtuuri 1

2.1 Backend-as-a-Service 2
2.2 Function-as-a-Service 3
2.3 BaaS:n ja FaaS:n suhde 4
2.4 Serverless-arkkitehtuurin hyödyt ja haitat 5
2.4.1 Kustannustehokkuus 5
2.4.2 Skaalautuvuus 5
2.4.3 Vastuunsiirto 6
2.4.4 Riippuvuus palveluntarjoajasta (”Vendor lock-in”) 7

3 Testisovelluksen määrittely 8

3.1 IBM Cloud 8
3.1.1 IBM Cloud Functions 8
3.1.2 Cloudant 9
3.2 Testisovellus 9

4 Toteutus 11

4.1 Tietokanta 11
4.2 FaaS-toiminnot 13
4.3 API 19
4.4 Feed ja Trigger 21
4.5 Front-end-sovellus 23
4.6 Monitorointi ja testaus 27
4.6.1 Monitorointi 27
4.6.2 Testaus 29

5 Arviointi 31

6 Yhteenveto 34

Lähteet 36

Lyhenteet

API	Application programming interface. Ohjelmointirajapinta, jonka mukaan sovellukset voivat kommunikoida keskenään.
REST	Representational State Transfer. HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen.
VPC	Virtual Private Cloud. Julkinen pilvialusta, johon voi luoda oman virtualisoidun yksityisen pilviympäristön.
SDK	Software Development Kit. Ohjelmistokehityspaketti, eli kokoelma ohjelmistokehityksen työkaluja yhdessä asennettavassa paketissa.

1 Johdanto

Suurimmalle osalle yrityksistä digitaalisuus on välttämätöntä. Mikäli nykyajan maailmassa haluaa menestyä, on yritysten kyettävä reagoimaan asiakkaiden tarpeisiin nopeasti ja kustannustehokkaasti. Tämän takia pilviteknologioiden ja -palveluiden käyttö sovelluskehityksessä on ollut nousussa tällä ja viime vuosikymmenellä. Pilviteknologiat ovat järkevän digitaalisen kehityksen edellytys. Pilvipalvelut tarjoavat ulkoistettavia sovelluskehityksessä tarvittavia palveluita ja niiden osia, jotka helpottavat kehitystä, mikä poistaa vastuuta kehittäjiltä ja siirtää sen palveluidentarjoajille.

Serverless on noussut pilvipalveluiden suosion myötä uudeksi kiinnostuksen kohteeksi. Serverless-teknologialla tarkoitetaan käytön perusteella maksettua palvelinkapasiteettia pilvipalvelussa. Nimityksellä Serverless viitataan siihen, että palveluntarjoaja hoitaa liikenteen jakamisen vapaille palvelimille, joten kehittäjän ei tarvitse huolehtia siitä. Monet isot yritykset, kuten Netflix, Reuters ja AOL käyttävät Serverless-arkkitehtuuria, ja sen käyttö on koko ajan lisääntyvää myös pienemmissä yrityksissä sekä sovelluksissa. [1.]

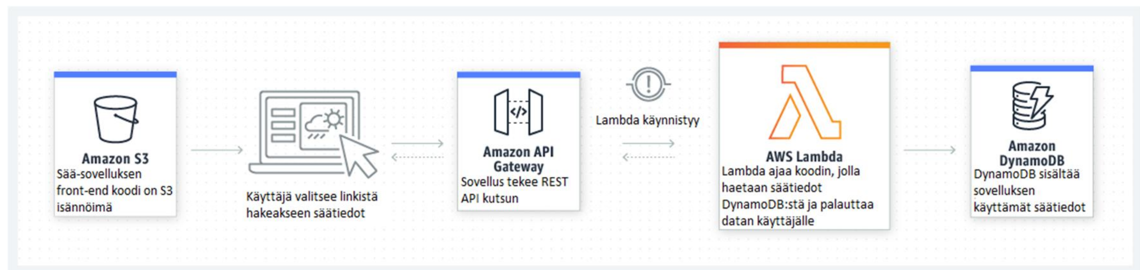
Insinöörityön tavoitteena oli perehtyä tarkemmin Serverless-arkkitehtuuriin ja sen hyötyihin sekä haittoihin sovelluskehityksessä. Työssä toteutettiin kustannusten seurantasovellus IBM Cloud -kehitysalustaa käyttäen. Testisovelluksen avulla tutkittiin, onko Serverless toimiva tapa pienen verkkosovelluksen kehityksessä ja onko IBM Cloud toimiva alusta Serverless-arkkitehtuuriin.

Testisovelluksen avulla saatiin selvitettyä, mitä kannattaa ottaa huomioon Serverless-arkkitehtuuria noudattaessa, onko tapa hyödyllinen sekä mistä pitää luopua, mikäli haluaa kehittää sovelluksia Serverless:in mukaan. Toteutuksen yhteydessä selvisi myös, kannattaako kehitystä tehdä IBM Cloud -alustaa käyttäen vai kannattaako harkita muita kehitysalustoja.

2 Serverless-arkkitehtuuri

Serverless-arkkitehtuuri kuvaa mallia, jossa sovellus käyttää pääosin kolmannen osapuolen tarjoamia pilvipalveluita, ja näin ulkoistaa vastuun palvelinpuolen logiikasta ja nii-

den prosesseista ulkoisille palveluidentarjoajille. Serverless-arkkitehtuurissa eliminoidaan infrastruktuurin hallinnointitehtäviä kehittäjiltä, kuten käyttöjärjestelmäpäivityksiä ja kapasiteetin määrityksiä, jolloin tavoitteena on vähentää yleiskustannuksia ja yksinkertaistaa prosesseja. [2; 3.]



Kuva 1. Serverless-arkkitehtuurin mukainen sääsovellus, kun käytetään Amazon Web Services (AWS) -tuotteita. Kuva perustuu AWS:n sivuilla olevaan esimerkkikuvaan. [3.]

Kuvan 1 mukaisesti Serverless-arkkitehtuurissa sovellus paloitellaan toiminnallisiin funktioihin, jota voidaan yksitellen kutsua ja skaalata tarpeen mukaan. [2.]

Arkkitehtuuri pitää sisällään kaksi eriävää, mutta päällekkäistä pilvipalvelumallia BaaS ja FaaS, joissa kummassakin resurssien hallinta on ulkoistettu kolmannelle osapuolelle. Serverless-sovelluksissa BaaS ja FaaS ovat molemmat usein käytössä. Monet suuret pilvipalveluiden tarjoajat, kuten esimerkiksi Amazon ja Google tarjoavatkin Serverless-portfolioita, jotka sisältävät molempien aihealueiden tuotteita. Kuvassa 1 on esimerkiksi käytössä BaaS-tuotteet Amazon API Gateway ja DynamoDB sekä FaaS-tuote AWS Lambda. [4.]

2.1 Backend-as-a-Service

BaaS kuvaa pilvipalvelumallia, jossa palveluidentarjoajat tarjoavat valmiita palvelinpuolen toimintoja, kuten todennusta, eli käyttäjän tai palvelun identiteetin varmentamista, tietokantojen hallintaa, tietovarastojen ylläpitoa tai muita palveluntarjoajan omia toimintoja. Näin sovellukseen ei tarvitse erikseen kehittää omia backend-toiminnallisuuksia, kun valmiisiin palveluihin voi integroitua. BaaS-tuotteita ovat esimerkiksi todennuspalvelut Auth0 ja AWS Cognito sekä pilvitietokantapalvelu Firebase. [5.]

Erona FaaS-malliin BaaS ei ole tapahtumapohjainen, vaan palveluita kutsutaan palveluntarjoajan tarjoaman API:n kautta tai käyttämällä tarjottuja SDK:ita. Kaikki BaaS-tuotteet eivät ole automaattisesti skaalautuvia, ellei palveluntarjoaja sitä erikseen tarjoa. [5.]

2.2 Function-as-a-Service

FaaS kuvaa pilvipalvelumallia, jossa kolmannen osapuolen pilvipalveluntarjoaja on vastuussa sille jaetusta toiminnosta, esimerkiksi koodin osasta, laskuttaen vain käytöstä. Koodi tyypillisesti ajetaan lyhytaikaisissa (”ephemeral”) ja tilattomissa konteissa (”stateless containers”), jotka ajetaan vain tapahtumapohjaisesti (”event-triggered”). Tapahtumat voivat olla esimerkiksi http-kutsuja, tietokantamuutoksia, tiedostojen latauksia tai seurantahälytyksiä. Palveluntarjoaja on myös vastuussa resurssien allokaatiosta ajettavalle toiminnolle, eli toiminto skaalautuu tarpeen mukaan. [6.]

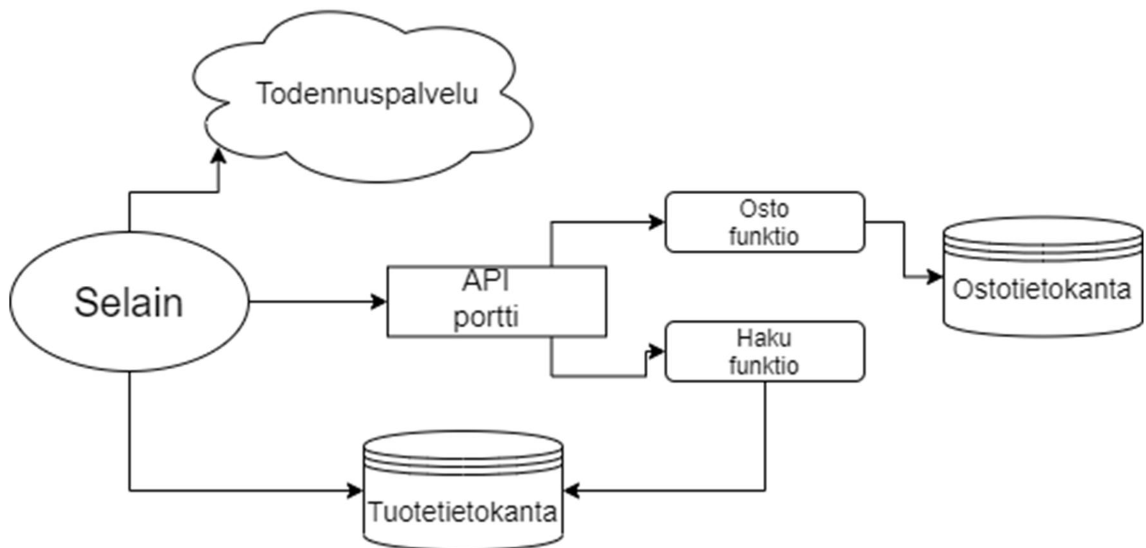
FaaS-funktiot kuvataan usein tilattomiksi, sillä niillä on merkittäviä rajoituksia, miten ne varastoivat dataa. Koska funktiot ovat lyhytaikaisia, ei ole mitään takeita siitä, että data, joka on tallennettu muistiin, pysyisi muistissa monen funktion ajon ajan. Jos funktion siis halutaan säilyttävän dataa, se pitää ulkoistaa funktioinstanssin ulkopuolelle, esimerkiksi tietokantaan. [4.]

Funktioiden lyhytaikaisuus rajoittaa myös niiden käyttömahdollisuuksia. Pitkään kestävät ajot eivät ole mahdollisia, vaan funktioita voi joutua ketjuttamaan, mikäli haluaa korvata pitkäkestoisen toiminnallisuuden. AWS on esimerkiksi määritellyt FaaS-funktion ajon kestoksi 300 sekuntia ja Google 500 sekuntia. [7.]

Käynnistysviiveeltä on mahdotonta välttyä, kun funktiot ovat tapahtumapohjaisia, eivätkä ole koko ajan ajossa. Viive voi vaihdella millisekunneista sekunteihin, mikä riippuu monesta eri tekijästä. Warm start viittaa tilanteeseen, jossa käytetään uudelleen valmiiksi luotua kontti-instanssia funktiosta, jolloin viive on lyhyt. Cold start taas tarkoittaa tilannetta, jossa joudutaan luomaan uusi kontti-instanssi funktiolle, ennen kuin se voidaan ajaa. Cold startien käynnistysviiveeseen vaikuttaa ohjelmointikieli, funktion koko, funktioiden ketjuttaminen sekä VPC:n käyttö. [4; 8.]

2.3 BaaS:n ja FaaS:n suhde

Aiemmin mainittiin, että molemmissa pilvipalvelumalleissa, BaaS:ssa ja FaaS:ssa ulkoistetaan resursseja palveluntarjoajalle. Ne ovat kuitenkin toiminnaltaan hyvin erilaisia. BaaS-palvelut on tarkoitettu hallinnoimaan backend-toiminnallisuutta, toisin kuin FaaS-palvelut, jotka ovat arkkitehtuurisesti tapahtumapohjaisia ja auttavat näin suorittamaan haluttuja tapahtumia. [9.]



Kuva 2. Yksinkertaistettu kuva Serverless-arkkitehtuuria noudattavasta sovelluksesta. Kuva perustuu englanninkieliseen lähteeseen. [4.]

Kuvassa 2 on yksinkertainen sovellus, jonka avulla voidaan havainnollistaa BaaS:n ja FaaS:n suhdetta toisiinsa sovelluskehityksessä. Koko backend-logiikka on hajautettu erillisiin toiminnallisuuksiin yhden ison palvelinpuolen logiikan sijaan. Todennuspalvelu, tietokannat ja API-portti ovat kaikki eri BaaS-palveluita, joita sovellus kutsuu erikseen. FaaS-osa sovellusta ovat haku- ja ostofunktiot, jotka käynnistyvät API-portin tietystä kutsusta. Nämä funktiot ovat sovelluskohtaisia ja suorittavat halutun toiminnon. Ne myös skaalautuvat kutsujen määrän mukaan. [4; 9.]

Yhteenvetona kaikki BaaS-palvelut ovat siis valmiita kolmannen osapuolen palveluita toiminnallisuuksineen, jotka voi ottaa käyttöön suoraan. FaaS-palveluissa koodi on kehittäjän tekemää sekä sovelluskohtaista, mutta sen ajo ja hallinnointi pysyvät palveluntarjoajalla. Näin saadaan arkkitehtuurimalli, jossa sovelluksen osilla on tietty toiminnallinen rooli, joka tekee sovelluksesta joustavamman ja helpomman muuttaa tai päivittää. [4; 9.]

2.4 Serverless-arkkitehtuurin hyödyt ja haitat

Serverless tarjoaa monia hyötyjä sovelluskehitykseen FaaS:in rajoitteista huolimatta. On tärkeää kiinnittää huomiota olennaisiin ominaisuuksiin, jotta sovelluskehittäminen olisi mahdollisimman helppoa ja mahdollisia riskejä kykenee mitätöimään.

2.4.1 Kustannustehokkuus

Pilvipalveluidentarjoajat veloittavat palveluistaan käytön mukaan. 'Pay-as-you-go'-malli mahdollistaa optimaalisen resurssien hyödyntämisen ilman, että joutuu maksamaan käyttämättömästä palvelinajasta. Mikäli palvelut ovat käytössä koko ajan, voi hinta kuitenkin nousta kalliimmaksi kuin tavallinen palvelimien ylläpito. [10.]

Amazon Web Services, lyhennettynä AWS, tarjoaa laskurin, jolla voi arvioida palveluiden hintoja etukäteen. Laskurissa on myös valmiita esimerkkejä palveluiden yhteenlasketuista hinnoista sovelluskäytössä. Esimerkiksi jos haluaisi ottaa käyttöön yksinkertaisen Node.js-web-sovelluksen, maksaisi se Free Tierin kanssa (Free Tier on AWS:n uusille asiakkaille tarjoama 12 kuukauden alennus) noin 26 Yhdysvaltain dollaria kuukaudessa. Kuitenkin isompien sovellusten kanssa hinta nousee. Jos haluaisi ottaa käyttöön ison web-sovelluksen, jonka kuukausittaiset kävijämäärät olisivat noin 100 000, maksaisi se noin 970 Yhdysvaltain dollaria kuussa. [12.]

2.4.2 Skaalautuvuus

Automaattinen skaalautuvuus tietoliikenteen mukaan on yksi Serverless:in suurista hyödyistä. Sovelluksen on mahdollista kasvattaa palvelinten käyttöä tarpeen vaatiessa tai pienentää sitä, mikäli tietoliikenne on odotettua vähäisempää. Kuten aiemmin mainittiin, skaalautuva sovelluksen koko vaikuttaa samalla kustannuksiin, joko niitä nostaen tai vähentäen. [10; 13.]

Skaalautuvuus on riippuvainen käyttäjien sijainnista ja verkkoyhteydestä. Suurimmilla palveluntarjoajilla onkin palvelinkeskuksia ympäri maailmaa, mikä vähentää viiveitä sekä mahdollistaa sovellusten toiminnan sijainnista riippumatta. Kuitenkin kaikissa palvelinkeskuksissa ei välttämättä ole tarjota samoja toiminnallisuuksia, mikä kannattaa huomioida sovelluksen kehityksessä sekä palveluntarjoajan valinnassa. Esimerkiksi Amazonin

palvelinkeskuksilla on erilaisia alueita (”Regions” ja ”Availability Zones”), joiden resurssit eroavat toisistaan. [10; 14.]

2.4.3 Vastuunsiirto

Niin kuin aiemmin on mainittu, Serverless-arkkitehtuurissa palveluntarjoaja on vastuussa infrastruktuurin hallinnoinnista ja ylläpidosta. Tämä säästää aikaa kehittäjiltä, jotka voivat keskittyä muihin kehitystehtäviin palvelinpuolen ylläpidon sijaan. [10.]

Pilvipalveluiden yleistyminen on tuonut mukanaan uusia käsitteitä, kuten X-as-a-Servicen, lyhennettynä XaaS. Kyseinen käsite kuvaa pilvipalveluiden eri palvelumalleja. Näistä tunnetuimpia ovat IaaS, PaaS ja SaaS. IaaS, eli Infrastructure-as-a-Service on malli, jossa palvelinsalien infrastruktuuri ulkoistetaan pilvipalveluntarjoajalle. Infrastruktuurin kaksi päätehtävää ovat tallennustilan ja laskentatehon tarjoaminen asiakkaille. Platform-as-a-Service, lyhennettynä PaaS, kuvaa mallia, jossa pilvipalveluntoimittaja tarjoaa palvelualustan asiakkaan käyttöön. SaaS, eli Software-as-a-Service on malli, jossa ohjelmisto tarjotaan palveluna perinteisen lisenssipohjaisen tavan ja ohjelmiston asentamisen sijaan. Esimerkiksi web-pohjaiset sähköpostipalvelut Outlook ja Hotmail ovat SaaS-palveluita.

Tunnetuimpien pilvipalvelumallien rinnalle on noussut myös aiemmin käsitellyt BaaS ja FaaS. Kaikissa palvelumalleissa vastuunsiirto on eriasteista. Kuva 3 havainnollistaa, mikä näissä eri palvelumalleissa on asiakkaan ja mikä toimittajan vastuulla. Kuvasta voi huomata, että FaaS-mallissa on suurin vastuu palveluntoimittajalla. [11.]

Itse hallinnoitava | Palvelun tarjoajan hallinnoima | Ei sovellettavissa

IaaS	PaaS	SaaS	FaaS
Funktiot	Funktiot	Funktiot	Funktiot
Sovellukset	Sovellukset	Sovellukset	Sovellukset
Data	Data	Data	Data
Ajoympäristö	Ajoympäristö	Ajoympäristö	Ajoympäristö
Tietovarasto	Tietovarasto	Tietovarasto	Tietovarasto
Virtualisaatio	Virtualisaatio	Virtualisaatio	Virtualisaatio
Laitteisto	Laitteisto	Laitteisto	Laitteisto

Kuva 3. Vertailu vastuun jaosta erilaisissa pilvipalveluiden palvelumalleissa. Kuva perustuu englanninkieliseen lähteeseen. [10.]

Vastuunsiirrossa on kuitenkin ongelmansa. Testaus ja virheenselvitys hankaloituvat, kun ei ole näkyvyyttä backend-prosesseihin ja ne on pilkottu erillisiin pieniin osiin. Riskiä voi vähentää ottamalla käyttöön palvelun testauksen mahdollistamisen. Uusia tietoturvariskejä voi syntyä, kun backend on ulkoistettu ja sinne lisätään henkilökohtaista tai arkaluonteista dataa. Palveluntarjoajien palvelimilla ajetaan useiden eri yritysten sovelluksia, ja mikäli palvelimia ei ole konfiguroitu oikein, voi tilanne johtaa datan altistumiseen. [13.]

2.4.4 Riippuvuus palveluntarjoajasta (”Vendor lock-in”)

Ulkoistamalla palvelinpuolen logiikan, sovellus tulee vääjäämättömästi riippuvaiseksi palveluntarjoajasta. Muutokset toimittajan palveluissa tai hinnoittelussa ovat mahdollisia ja voivat vaikuttaa sovellukseen. Vaihto palveluntarjoajasta toiseen voi olla hankalaa ja vaatia sovelluksen muuttamista, sillä kaikki palvelut eivät välttämättä ole samanlaisia. [10.]

3 Testisovelluksen määrittely

Toteutettavan testisovelluksen kuuluu olla sellainen, joka mittaa Serverless-arkkitehtuurin ominaisuuksia, joten tarvitaan kehitysalusta, joka mahdollistaa sen käytön. Valittu alusta tarjoaa monia erilaisia BaaS- ja FaaS-palveluja, joiden avulla sovellus tehdään.

3.1 IBM Cloud

IBM Cloud on pilvialusta, joka tarjoaa monia erilaisia palveluita esimerkiksi datan hallintaan, tekoälyyn sekä esineiden internetiin liittyen. IBM Cloud sisältää IaaS-, SaaS- ja PaaS-palvelumalleja, palvelujen mukaan. Testisovelluksessa käytetään IBM Cloudin tarjoamia palveluita Cloud Functions, joka on PaaS-palvelu, sekä Cloudant, joka on BaaS-palvelu.

3.1.1 IBM Cloud Functions

IBM Cloud Functions perustuu avoimen lähdekoodin Apache OpenWhisk -kehitysalustaan, joka hallinnoi infrastruktuurin, palvelimet ja skaalautuvuuden käyttämällä kontteja. Kontit ovat standardisoituja ohjelmistoja, jotka pakkaavat koodin ja sen riippuvuudet, jotta sovellusta voidaan ajaa, ja siirtää helposti eri ympäristöissä. [15; 16.]

Cloud Functions sekä OpenWhisk tukevat ohjelmointimallia, jossa kehittäjät kirjoittavat toimintoja (”Action”) haluamallaan ohjelmointikielellä. Action on OpenWhiskin käsite FaaS-funktiolle. Käytetään niistä suomenkielistä nimitystä toiminto. Toiminnot on mahdollista asettaa ketjuun (”Sequence”), joka mahdollistaa monien toimintojen kutsun peräkkäin sekä yksittäisten toimintojen uudelleenkäytön. Ketjussa olevat toiminnot kutsutaan järjestyksessä, jossa edellisen toiminnon vastaus välittyy seuraavaksi kutsuttavalle toiminnolle. Ketjut tai toiminnot ajetaan tiettyjen tapahtumien (”Triggers”) pohjalta, jotka voivat olla syötteitä ulkoisista lähteistä (”Feeds”) tai esimerkiksi http-kutsuja. Tarjolla on paljon valmiita palveluintegraatiomahdollisuuksia, esimerkiksi testisovelluksessa käytettävään tietokantapalveluun Cloudant ja sekä viestipalveluun Slack. [15; 17.]

3.1.2 Cloudant

Cloudant perustuu avoimen lähdekoodin Apache CouchDB:hen, joka on dokumenttiorientoitunut tietokantajärjestelmä. Toisin kuin relaatiotietokanta, CouchDB:n tietokanta sisältää kokoelman erillisiä JSON-dokumentteja, joissa jokaisessa on oma skeemansa. Tietokantaan kommunikointi onnistuu http-kutsujen avulla. [18; 19.]

Koska Cloudant pohjautuu CouchDB:hen, ovat niiden API:t lähes täysin yhteensopivat. Täysin hallinnoidussa Cloudant:issa on kuitenkin eroja CouchDB:hen. Esimerkiksi todennus mahdollisuuksia on erilaisia, kuten API-avainten käyttö, jossa generoidulla avaimella on oikeus päästä tietokantaan, muttei oikeutta tietokannan hallinnointityökaluun. [20.]

3.2 Testisovellus

Jotta Serverless-arkkitehtuuria voidaan tutkia, määritellään testisovellus, joka toteutetaan valitussa testiympäristössä. Tarkoituksena on toteuttaa yksinkertainen kustannusten seuranta web-sovellus, jossa voidaan käyttöliittymän kautta lisätä tai poistaa kustannuksia päiväkohtaisesti sekä saada ilmoitus ylittyneestä kuukausittaisesta kulutusrajasta.

Taulukko 1. Kustannusten seurantasovelluksen vaatimusmäärittely.

Vaatimus	**Prioriteetti**
Sovellus käyttää REST API:a, ja vastaa http-kutsuihin: GET/POST/DELETE	1
Sovellus tallentaa ja hakee arvoja tietokannasta.	2
Sovellus reagoi tietokantamuutokseen lähettämällä sähköpostin, mikäli kuukausittainen kulutusraja ylittyy.	3
Sovellus skaalautuu tarpeen mukaan.	4
Sovellusta kykenee testaamaan.	5
Sovellusta kykenee monitoroimaan.	6

Taulukossa 1 on sovelluksen vaatimusmäärittely. Taulukon prioriteettimäärityksellä asetetaan vaatimukset tärkeysjärjestykseen, jotta vertaillessa voidaan tutkia, pystyikö kehitysympäristöllä toteuttamaan kaikki vaatimukset, ja missä tärkeysjärjestyksessä.

Sovellukselle määriteltiin API ennen toteutusta, jotta toteutus olisi selkeämpää. API on kuvattu taulukossa 2. REST API määriteltiin toiminnallisten tarpeiden mukaan. Se on englanniksi, sillä se kuuluu hyviin REST API -kehitystapoihin.

Taulukko 2. Sovelluksen REST API:n määritelmä

HTTP metodi	**Polku**	**Toiminto**
GET	/expenses/totals/year	Hae kuukausittaiset kustannukset
GET	/expenses/totals/month	Hae päivittäiset kustannukset
POST	/expenses/entries	Lisää kustannus päivälle
POST	/expenses/limits	Kuukausittainen kulutusraja
DELETE	/expenses/totals/day	Poista kustannus päivältä

Kaksi GET-hakua hakevat yhteenlasketut kustannukset kuukausittain sekä päivittäin. Parametrina kuukausittaiselle haulle annetaan haluttu vuosi, päivittäiselle haulle annetaan haluttu vuosi sekä kuukausi. POST-kutsuille ei anneta parametreja vaan JSON-objektit, joka sisältävät tietokantaan tallennettavat tiedot. DELETE-kutsulle annetaan parametrina vuosi, kuukausi ja päivä.

Sovelluksen käyttämään tietokantaan pitää tallentaa kustannustietoina vuosi, kuukausi, päivä ja summa. Kuukausittaisen kulutusrajan tallentamiseksi tiedoiksi tarvitaan kulutusraja, vuosi, kuukausi sekä sähköpostiosoite. Tiedot tallennetaan JSON-objekteissa, esimerkkinä esimerkkikoodi 1, jossa on yksi kustannustieto.

```
{
     "_id": "597a97b05a2ac0c45219b3fd02817e8f",
     "_rev": "1-d3339d62d69e4f372619d9289f509f95",
     "day": "31",
     "expense": 12.54,
     "month": "01",
     "year": "2020"
}
```

Esimerkkikoodi 1. Tietokantaan tallennettu JSON-objekti, joka sisältää kustannustiedot sekä uniikit, automaattisesti generoidut arvot id ja rev.

Esimerkkikoodin 1 mukaisesti päiväystiedot annetaan merkkijonoina ja kustannustieto numeroarvona helpottamaan kustannusten käsittelyä, kuten yhteenlaskua. Kuukausi ja päivä annetaan aina kaksimerkkisinä yhdenmukaisuuden takia.

4 Toteutus

Ennen toteutuksen aloitusta on rekisteröidyttävä ja luotava tunnus IBM Cloudiin. Ilmainen Lite-tunnus ei vaadi luottokorttitietoja rekisteröityessä eikä ole määräaikainen. Sillä saa käyttöön rajallisen määrän IBM:n palveluita, joilla pääsee tutustumaan tuotteisiin. Esimerkiksi Cloudant-palvelusta saa yhden gigatavun muistia ilmaiseksi käyttöön, joka riittää pienen testisovelluksen tekoon. Lite-tunnus on rajoitettu yhteen alueeseen ("Region"), joka on tässä tapauksessa Iso-Britannia ("eu-gb").

IBM Cloud tarjoaa kehitykseen käyttöliittymän sekä komentotyökalun. Käyttöliittymä on selainpohjainen. Komentotyökalut ovat asennettava erikseen valmiiksi määritellyllä asennuskomennolla, joka asentaa seuraavat ohjelmat.

- Git
- Docker
- Helm
- kubectl
- IBM Cloud Functions lisäosa
- IBM Cloud Object Storage lisäosa
- IBM Cloud Container Registry lisäosa
- IBM Cloud Kubernetes Service lisäosa
- Homebrew (Vain Mac-tietokoneille)
- curl (Vain Linux-käyttöjärjestelmille).

Toteutuksessa käytettiin Windows-käyttöjärjestelmää, joten Homebrew'ta ja curlia ei asennettu. Jotta Docker for Windows on mahdollista asentaa, Windows-versio pitää olla joko Pro tai Enterprise.

4.1 Tietokanta

Toteutus aloitettiin luomalla Cloudant-palveluinstanssi IBM Cloud -käyttöliittymän avulla. Ilmainen Lite-tunnus määritti valittavan hinnoittelusuunnitelman ja alueen. Instanssin nimeksi annettiin expenses-db. Todennustavaksi valittiin vaihtoehto: 'Use both legacy credentials and IAM', jossa palvelu generoi sekä legacy- että IAM-tunnukset. IAM eli Identity and Access Management -tunnukset määrittävät rooleittain käyttöoikeuksien määrän, kun legacy-tunnukset käyttävät http-todennusta, joka ei ole määritelty tietyille käyttäjille.

Koska tietokantaan haluttiin kutsua http-todennuksen avulla, käytettiin molempia todennustapoja, eikä vain suositeltua IAM-todennusta.

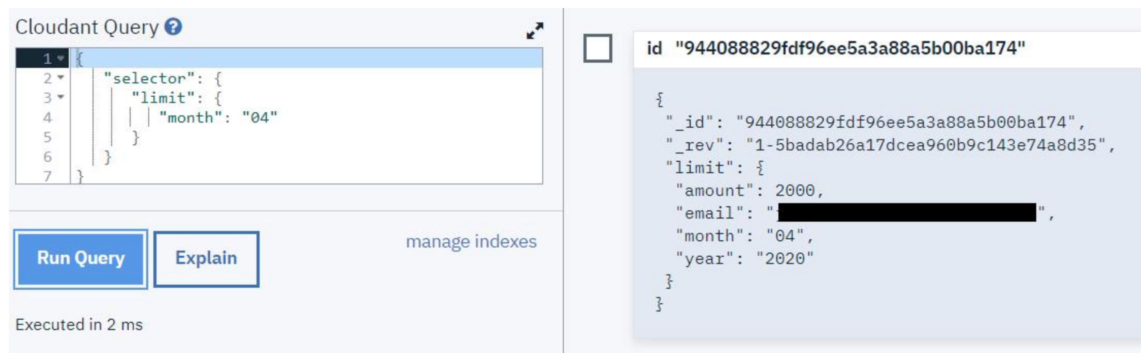
```
Windows PowerShell
PS C:\> ibmcloud resource service-instances
Retrieving instances with type service_instance in all resource groups in all locations under account Johanna T's Account as ...
OK
Name          Location   State    Type
expenses-db   eu-gb      active   service_instance
PS C:\> ibmcloud resource service-instance expenses-db
Retrieving service instance expenses-db in all resource groups under account Johanna T's Account as
m...
OK

Name:                  expenses-db
ID:                    crn:v1:bluemix:public:cloudantnosqldb:eu-gb:a/0841f849ebdc41efa135c6c8275d6ae5:c107d237-0948-4456

GUID:                  c107d237-0948-4456-
Location:              eu-gb
Service Name:          cloudantnosqldb
Service Plan Name:     lite
Resource Group Name:   Default
State:                 active
Type:                  service_instance
Sub Type:
Created at:            2020-02-27T15:03:03Z
Created by:
Updated at:            2020-02-27T15:03:36Z
Last Operation:
                       Status       create succeeded
                       Message      Provisioning is complete
                       Updated At   2020-02-27 15:03:36.216463717 +0000 UTC

PS C:\>
```

Kuva 4. Komentoliittymän näkymä luodusta Cloudant-instanssista, jossa näkee instanssin tiedot. Salaiset tiedot on piilotettu.

Palveluinstanssin luonnin jälkeen varmistettiin sen olevan aktiivinen komentoliittymän kautta. Kuvasta 4 näkee, että luodun instanssin tila on aktiivinen. Sen jälkeen käynnistettiin uusi Cloudant-työpöytä, jossa luotiin uusi tietokanta. Tietokannan nimeksi annettiin expenses.

Jotta tietokannasta haku olisi mahdollisimman tehokasta, käytettiin tietokantakyselyissä IBM Cloud Querya, joka on deklaratiivinen JSON-kyselysyntaksi Cloudant-tietokannoille. Tietokantaan voi luoda indeksejä, joiden mukaan tietokanta järjestää datan kyselyiden nopeuttamiseksi. Tietokantaan laadittiin kolme JSON-tyyppistä indeksiä helpottamaan kustannusten sekä kuukausittaisten kustannusrajojen hakua. Indekseille annettiin lista kenttien nimistä, joiden mukaan tietokanta indeksoidaan. Esimerkiksi kuukausittaisten kustannusrajojen indeksin kentäksi annettiin limit.month, jotta tietokannasta löydetään helposti dokumentit, jossa limit-objektin sisällä on kenttä month. Tietokannasta hakuindeksiä käyttämällä vaati tietynlaisen hakuobjektin. Tietokantakutsussa tulee olla selec-

tor-kenttä, joka määrittää, mitä dokumentteja haetaan. Esimerkiksi kuukausittaisen kustannusrajan haussa määriteltäisiin, että limit-objektin sisällä tulee olla month-kenttä tietyllä kuukausiarvolla.



Kuva 5. Käyttöliittymästä kuva, jossa vasen puoli näyttää tietokantakutsun, jossa haettiin kuukausittaista kulutusrajaa huhtikuulle ja oikealla puolella näkyy kutsun vastaus.

Kuvasta 5 voi nähdä kutsun keston, joka oli vain 2 millisekuntia. Indeksien teko tietokannalle ei ole pakollista, mutta viisasta, kun halutaan kustannustehokkaita ja nopeita toimintoja. Esimerkiksi ilman indeksiä tietokantahaku kesti 4 millisekuntia. Indeksin avulla hakuaika siis puolittui, jolla on suuri merkitys, jos tietokannassa on paljon dokumentteja.

4.2 FaaS-toiminnot

Node.js valittiin toimintojen ajoympäristöksi. Node.js on alustariippumaton avoimen lähdekoodin JavaScript-ajoympäristö, joka mahdollistaa koodin suorittamisen suoraan palvelimella. Toimintoja kykeni koodaamaan lokaalisti editorilla, jonka jälkeen ne komentoliittymän avulla määriteltiin toiminnoiksi tai kehittämään niitä suoraan selainkäyttöliittymässä. Kaikki toiminnallisuus toteutettiin ketjuttamalla yksittäisiä toimintoja kokonaisuuden muodostamiseksi.

Ensimmäinen toteutettava toiminnallisuus oli kustannuksen tallennus tietokantaan. Aloitettiin luomalla toiminto prepare-for-save, joka tarkisti, että kutsussa on kaikki tarpeelliset parametrit, palauttaen joko virheen tai JSON-objektin sisältäen tietokantaan tallennettavat tiedot formatoituna.

Tietokantaan tallennukseen käytettiin valmista Cloudant-pakettia, joka sisältää luku-, kirjoitus-, päivitys- ja poistotoiminnot. Aikaisemmin luotu Cloudant-instanssi voitiin yhdistää

valmiiseen pakettiin luomalla pakettisidonta (”package binding”). Sidonnan avulla palvelulle ei tarvitse välittää käyttöoikeuksia joka kutsun yhteydessä, vaan ne välittyivät suoraan palvelulle valmiin paketin toimintojen parametreissa. Tehdyn sidonnan jälkeen komentoliittymästä voitiin varmistaa kuvan 6 mukaisesti, että käyttöoikeudet välittyivät oikein vertaamalla Cloudant-käyttöoikeuksia ja valmiin binding-for-expenses-paketin parametreja.

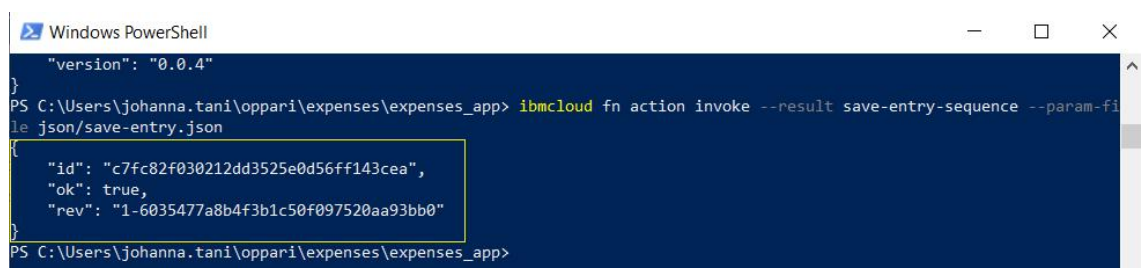
```
Windows PowerShell
PS C:\> ibmcloud resource service-key "Service credentials-1"
Retrieving service key Service credentials-1 in all resource groups under account Johanna T's Account as
...

Name:          Service credentials-1
ID:
Created At:    Thu Feb 27 15:06:41 UTC 2020
State:         active
Credentials:
               apikey:
               host:                                       5a17e-bluemix.cloudantnosqldb.appdomain.cloud
               iam_apikey_description:
               iam_apikey_name:
               iam_role_crn:
               iam_serviceid_crn:

               password:
               port:
               url:

               username:                                   7e-bluemix

PS C:\> ibmcloud fn package get binding-for-expenses parameters
ok: got package binding-for-expenses, displaying field parameters
[
    {
        "key": "bluemixServiceName",
        "value": "cloudantNoSQLDB"
    },
    {
        "key": "username",
        "value": "                                  7e-bluemix"
    },
    {
        "key": "host",
        "value": "                              5a17e-bluemix.cloudantnosqldb.appdomain.cloud"
    },
    {
        "key": "dbname",
        "value": "expenses"
    },
    {
        "key": "apihost",
        "value": "eu-gb.functions.cloud.ibm.com"
    },
    {
        "key": "password",
        "value": "                                                                 "
    }
]
```

Kuva 6. Komentoliittymässä näkee ensin haetut Cloudant-instanssin käyttöoikeudet sekä sen jälkeen haetut sidotut käyttöoikeudet pakettiin binding-for-expenses. Salaiset tiedot on piilotettu.

Kun toiminnot olivat valmiit, luotiin save-entry-sequence-ketju, johon liitettiin molemmat toiminnot. Ketjua kutsuttaessa ensin kutsuttiin prepare-for-save-toimintoa, jonka jälkeen binding-for-expenses-paketin tallennustoimintoa create-document. Mikäli dokumentin

teko ja tallennus onnistuu, Cloudant palauttaa JSON-objektin, jossa on tallennetun dokumentin id, tarkastusmerkkijono sekä onnistumiseen viittaava avain-arvopari ”ok:true”. Virhetilanteessa palautetaan kuvaus tapahtuneesta virheestä.

Valmista ketjua voitiin kutsua komentoliittymältä tai käyttöliittymältä. Komentoliittymältä kutsuttaessa voidaan tarkentaa, miten ketjuja tai toimintoja kutsutaan. Kutsuun voi antaa parametrin --blocking, joka jaksottaa kutsun ja odottaa toiminnon vastausta ja palauttaa sen ennen aikakatkaisua. Aikakatkaisu on 60 sekuntia, mikäli sitä ei ole erikseen määritelty. Ilman --blocking-parametria ketjun tai toiminnon kutsuminen palauttaa vain aktivaatiotunnuksen, ja toteutus jää taustalle. Vastauksen voi siinä tapauksessa katsoa myöhemmin aktivaatiotunnuksen avulla. Toinen hyödyllinen parametri on --result, joka tekee kutsun jaksottaisena, eli jää odottamaan vastausta ja palauttaa vain kutsutun toiminnon sisällön. Kutsuun ei erikseen tarvitse lisätä --blocking-parametria, kun toimintoa tai ketjua kutsutaan --result-parametrilla.



Kuva 7. Komentoliittymässä dokumentin tallennusta kutsuttu --result-parametrin kanssa. Keltaisella huomioitu toiminnon vastaus.

Parametrien erona on näytettävän tiedon määrä. Parametrilla --result-kutsu on hyödyllinen, kun tarvitaan vain kutsutun toiminnon vastauksen sisältö. Kutsu --blocking-parametrilla palauttaa paljon metatietoa kutsusta, joka voi olla hyödyllistä. Mukana palautuu myös toiminnon loki, joka näyttää lokille asetetut tiedot. Vertailemalla kuvia 7 ja 8, jossa on molemmissa kutsuttu samaa dokumentin tallennus ketjua, voidaan nähdä --result- ja --blocking-parametrien vastauksien eroavaisuudet.

Käyttöliittymältä tehdyt kutsut käyttävät aina parametria --blocking.

Windows PowerShell


```
}
PS C:\Users\johanna.tani\oppari\expenses\expenses_app> ibmcloud fn action invoke --blocking save-entry-sequence --param-file json/save-entry.json
ok: invoked /_/save-entry-sequence with id 93877b2a5c174daa877b2a5c176daa6a
{
    "activationId": "93877b2a5c174daa877b2a5c176daa6a",
    "annotations": [
        {
            "key": "topmost",
            "value": true
        },
        {
            "key": "path",
            "value": "/save-entry-sequence"
        },
        {
            "key": "kind",
            "value": "sequence"
        },
        {
            "key": "limits",
            "value": {
                "concurrency": 1,
                "logs": 10,
                "memory": 256,
                "timeout": 60000
            }
        }
    ],
    "duration": 509,
    "end": 1586068033259,
    "logs": [
        "c229ab3028094d87a9ab302809ad87f2",
        "1bb567285bff4f4ab567285bff5f4a1d"
    ],
    "name": "save-entry-sequence",
    "namespace": "",
    "publish": false,
    "response": {
        "result": {
            "id": "2958b9480bd42437c2aa4bf037cd8b64",
            "ok": true,
            "rev": "1-6035477a8b4f3b1c50f097520aa93bb0"
        },
        "size": 94,
        "status": "success",
        "success": true
    },
    "start": 1586068032616,
    "subject": "",
    "version": "0.0.4"
}
```


Kuva 8. Komentoliittymästä dokumentin tallennus ketjua kutsuttu --blocking-parametrilla. Keltaisella huomioitu aktivaatiotunnus ja kutsutun ketjun vastaus. Salaiset tiedot on piilotettu.

Jotta tietokantaan tallennusketjua pystyi kutsumaan verkon tai API:n kautta, piti ketjulle sallia http-kutsut. Toimintoja, jotka reagoivat http-tapahtumiin, kutsutaan verkkotoiminnoiksi (”Web Action”). Niitä voi kutsua ilman todennusta ja niiden on palautettava JSON-objekti, jossa on arvoina ylätunnus, statuskoodi sekä ohjelmakoodivastaus. Oletuksena ylätunnus on tyhjä, ja mikäli ohjelmakoodivastaus ei ole tyhjä, niin statuskoodi on 200. Muuten statuskoodi on 204, ei sisältöä. Http-kutsujen sallinta tehtiin vain save-entry-sequence-ketjulle, joten ketjun yksittäisiä toimintoja ei voi erikseen kutsua verkon välityksellä.

Kuukausittaisen kulutusrajan tallennus tietokantaan on hyvin samanlainen kuin kustannuksen tallennus. Ensimmäinen toiminto tarkistaa, että kutsussa välittyy testisovelluksen suunnittelussa määritetyt oikeat tiedot ja palauttaa virheen, mikäli ei ole.

```
function main( params ) {
     if ( !params.year || !params.month ) {
          return Promise.reject({ error: 'Date data missing!'});
     }
     if ( !params.limit || !params.email  ) {
          return Promise.reject({ error: 'Limit or email data missing!'});
     }
     return {
          doc: {
               limit: {
                    amount: params.limit,
                    month: params.month,
                    year: params.year,
                    email: params.email
               }
          }
     };
}
```

Esimerkkikoodi 2. Toiminnossa prepare-limit-save varmistetaan, että kutsussa annetaan tarvittavat tiedot ja palautetaan ne seuraavalle toiminnolle.

Esimerkkikoodi 2:sta näkee, että tiedot palautetaan doc-objektin sisällä. Doc-objekti on tarpeellinen, sillä Cloudant-paketin tietokantaan tallennustoiminto tallentaa doc-objektin sisältämät tiedot. Toisena toimintona käytettiin samaa Cloudant-paketin valmista toimintoa create-document. Nämä lisättiin ketjuun save-limit-entry-sequence, ja ketjun kutsunta verkon välityksellä sallittiin.

Tietokannasta haku toiminnallisuudet toteutettiin hyvin samalla tavalla kuin tallennuskin. Tietokannasta tarvitsi hakea koko vuoden tiedot kuukauden tarkkuudella sekä yksittäisen kuukauden tiedot päivän tarkkuudella. Toiminnollisuuksien tekeminen aloitettiin luomalla toiminnot, jotka tarkistavat annetun datan, mutta jos vuosi tai kuukausi tieto puuttui, asetettiin kuluvan vuoden ja kuukauden arvot. Toimintojen vastaukset formatoitiin noudattamaan samaa muotoa kuin tietokannan hakuobjekti.

Tietokannasta hakuun käytettiin molemmissa toiminnollisuuksissa Cloudant-paketin valmista toimintoa exec-query-find. Vastauksessa palautui lista löydetyistä dokumenteista sekä niin sanottu kirjanmerkkitieto, jonka avulla pystyisi hakemaan lisää dokumentteja, jos kaikkia dokumentteja ei olisi palautettu. Kirjanmerkkitieto palautetaan aina, vaikka kaikki dokumentit olisi palautettu haun yhteydessä.

Palautetuissa listoissa on tietoa, jota oli turha palauttaa kutsun sisällössä ja jota piti formatoida, joten luotiin toiminnot datan käsittelylle. Toiminnoissa jätettiin pois tietokannasta palautetut id- ja rev-kentät ja laskettiin yhteen tietyn päivän ja kuukauden summat, jotta lopputuloksena vastauksessa oli kuukauden tai päivän yhteenlaskettu summa, sekä päivä tai kuukausi. Esimerkkikoodi 3:ssa on kuvattuna kuukausittaisten kustannusten formatointitoiminto, joka havainnollistaa tarkemmin, miten kustannukset laskettiin yhteen joka kuukaudelle.

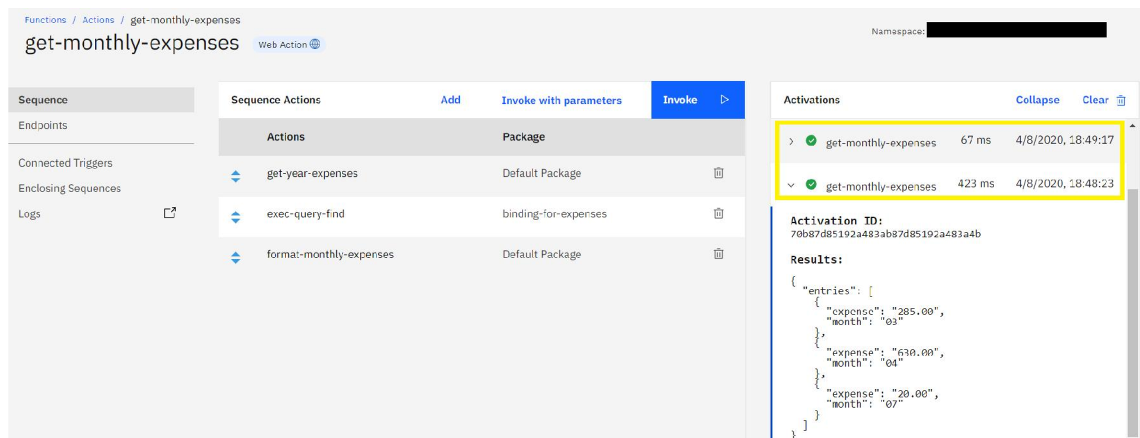
```
function main(params) {
	var monthArray = [];
	var objectArray = [];

	params.docs.map((row) => {
		if(!monthArray.includes(row.month)) {
			monthArray.push(row.month);
			let obj = {month: row.month, sum: parseFloat(row.expense)};
			objectArray.push(obj);
		} else {
			let objIndex = objectArray.findIndex((obj => obj.month ==
row.month));
			objectArray[objIndex].sum += parseFloat(row.expense);
		}
	})
	return {
		entries: objectArray.map(x => { return {
			month: x.month,
			expense: roundSum(x.sum)
		}})
	};
}

function roundSum(sum) {
    let rounded = parseFloat(sum).toFixed(2);
    return rounded.toString();
}
```

Esimerkkikoodi 3. Toiminto formatoi haetut kuukausittaiset kustannukset. Main-funktiossa asetetaan tietokannasta haetut tiedot doc-listasta uuteen listaan objectArray. Lista monthArray toimii tarkistuksena, onko kyseinen kuukausimerkintä jo lisätty objectArray-listaan. Jos monthArray-listassa on jo kyseinen kuukausi, summataan kuukauden kustannukset yhteen. Funktio roundSum pyöristää kustannuksen kahden desimaalin tarkkuudella.

Samoin kuin muissakin toiminnallisuuksissa toiminnot liitettiin verkon kautta kutsuttaviin ketjuihin get-monthly-expenses, joka haki annetun tai nykyisen vuoden kuukausittaiset kustannukset ja get-daily-expenses, joka haki annetun tai nykyisen kuukauden päivittäiset kustannukset.

Ketjun get-monthly-expenses toiminnallisuutta varmistaessa voi nähdä käynnistysviiveen eron. Kuva 9 havainnollistaa, kuinka ensimmäistä kertaa kutsuttaessa ketjun suo-

rittaminen kesti 423 millisekuntia, kun toiminnoille luodaan kontti-instanssi, jossa ne ajettiin. Toisen kerran kutsuttaessa ajo kesti enää vain 67 millisekuntia, kun kontti-instanssi oli jo valmiina.



Kuva 9. IBM Cloud Functions -käyttöliittymänäkymä, jossa ajettu ketju kahteen kertaan havainnollistamaan FaaS-funktioiden käynnistysviivettä.

Kustannusten poisto toteutettiin niin, että sovelluksesta on mahdollista poistaa vain koko päivän kustannukset. Kustannusten poistoon tehtiin ketju delete-day-expenses-sequence, johon liitettiin toiminto get-daily-expense ja jo aikaisemmin käytetty tietokantatoiminto exec-query-find, jotka ketjutettuna hakivat pyydetyn päivän kustannukset. Tietokannasta poisto tapahtuu Cloudantin bulkkioperaatiolla (”Bulk operations”), jossa voidaan käsitellä montaa dokumenttia samanaikaisesti yhdellä kutsulla. Kutsuun liitetään poistettavien dokumenttien id- ja revisio-arvo sekä avain-arvopari ”_deleted:true”. Ennen Cloudant-paketin valmiin toiminnon kutsuntaa toteutettiin datan formatointi toiminto, joka lisäsi avain-arvoparin vastaukseen. Dokumenttien poiston onnistuessa palautetaan vastauksessa poistettujen dokumenttien id- ja revisioarvot.

4.3 API

Ilmaiselle Lite-tunnukselle on saatavana IBM Cloudin tarjoama Legacy API Gateway palvelu, johon ei tarvitse luoda erillistä instanssia. Se myös tarkoittaa, ettei itse palvelua voi hallinnoida. Palvelun avulla voi kuitenkin luoda, suojata ja jakaa API:ja.

IBM Cloud Functions käyttöliittymän APIs-sivulta luodaan API, jotta luotuja FaaS-ketjuja voidaan kutsua sen kautta. API:lle annetaan nimeksi expenses ja asetetaan se sovelluksen REST API määrittelyn mukaisesti myös API-polun pääksi. Operaatiot luotiin myös aiemmin määritellyn REST API -määrityksen mukaan. Kuvasta 10 näkee Cloud Functionsin API-sivun, jossa on määriteltynä kaikki API-operaatiot.

API basics
Specify a descriptive name and a base path for this API.

API name
expenses

Base path for API
/expenses

Operations
Create API operations that invoke Cloud Functions actions.

Create operation +

Path	Verb	Package	Action
/totals/year	GET	default	get-monthly-expenses
/totals/month	GET	default	get-daily-expenses
/limits	POST	default	save-limit-entry-sequence
/entries	POST	default	save-entry-sequence
/totals/day	DELETE	default	delete-day-expenses-sequence

Kuva 10. Käyttöliittymässä listaus API-operaatioista. Kuvassa näkyy API-polku, http-metodi sekä kutsuttavan ketjun nimi.

API:lle ei ole mahdollista määritellä omaa verkkotunnusta ilmaisella Lite-käyttäjällä. Aina kun API kytketään päälle, sille muodostuu uusi verkko-osoite. Tämän voi tietty välttää siten, ettei koskaan kytke API:a pois päältä, mutta se on ylläpidon ja kehityksen kannalta hankalaa.

```
PS C:\Users\johanna.tani\oppari\expenses\expenses_app> curl https://bdd671d8.eu-gb.apigw.appdomain.cloud/expenses/totals/month?month=04

StatusCode        : 200
StatusDescription : OK
Content           : {
                      "entries": [{
                        "day": "01",
                        "expense": "10.00"
                      }, {
                        "day": "04",
                        "expense": "50.00"
                      }, {
                        "day": "15",
                        "expense": "600.00"
                      }, {
                        "day": "11",
                        "expense": "10.00"
                    ...
RawContent        : HTTP/1.1 200 OK
                    Connection: keep-alive
                    X-Request-ID: c366835008f901e3917d6a5ddf6dd1f1,jvCcqu4c6ghFsUXP8Ec9KpnOAn5Vk7aY,jvCcqu4c6ghFsUXP8Ec
                    9KpnOAn5Vk7aY
                    Access-Control-Allow-Origin: *
                    Access-Contro...
Forms             : {}
Headers           : {[Connection, keep-alive], [X-Request-ID, c366835008f901e3917d6a5ddf6dd1f1,jvCcqu4c6ghFsUXP8Ec9KpnO
                    An5Vk7aY,jvCcqu4c6ghFsUXP8Ec9KpnOAn5Vk7aY], [Access-Control-Allow-Origin, *], [Access-Control-Allow
                    -Methods, GET, POST, PUT, DELETE, PATCH, HEAD, OPTIONS]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 206
```

Kuva 11. Kuvassa komentoliittymältä haettu huhtikuun kustannuksia API:n kautta.

Kuvan 11 mukaisesti, komentoliittymältä kutsuttaessa voi varmistaa API:n toimivan oikein. Vastauksesta voi tarkastaa statuskoodin, ylätunnukset sekä osan sisällöstä.

4.4 Feed ja Trigger

Jotta haluttu toiminnallisuus ilmoituksesta, jos kuukausittainen kulutusraja ylittyy, saatiin toteutettua, piti hyödyntää IBM Cloud Functionsin toiminnallisuuksia Feed ja Trigger. Feed, suomeksi syöte, on tapahtumajono, jonka jokainen tapahtuma kuuluu samalle laukaisijalle (”Trigger”). Laukaisija ilmoittaa, että tietyn tyyppiseen tapahtumaan pitää reagoida. Tapahtumat voivat olla käyttäjiltä tai muista lähteistä. Syötteiden avulla ulkoisista lähteistä tulevat tapahtumat voivat laukaista Cloud Functions -toimintoja tai ketjuja. [17.]

Tietokantamuutoksen reagointiin käytettiin Cloudantin valmiista paketista muutossyötettä, joka laukaisee tapahtuman joka kerta, kun tietokantaan lisätään tai päivitetään dokumenttia. Ensin luotiin laukaisija, joka ilmoittaa expenses-tietokannan muutossyötteen tapahtumista. Laukaisijan nimeksi annettiin monthly-limit-check-trigger. Valmiin pakettisidonnan takia ei tarvinnut erikseen konfiguroida käyttöoikeuksia tietokantaan.

Seuraavaksi tehtiin ketju ja sen toiminnot, jotka toteutuksen jälkeen yhdistettiin laukaisijaan. Ensimmäinen toiminto ketjuun check-monthly-limit-sequence oli jo toteutettu get-month-expenses-toiminto, jota käytettiin tietokannan haussa. Seuraava toiminto oli

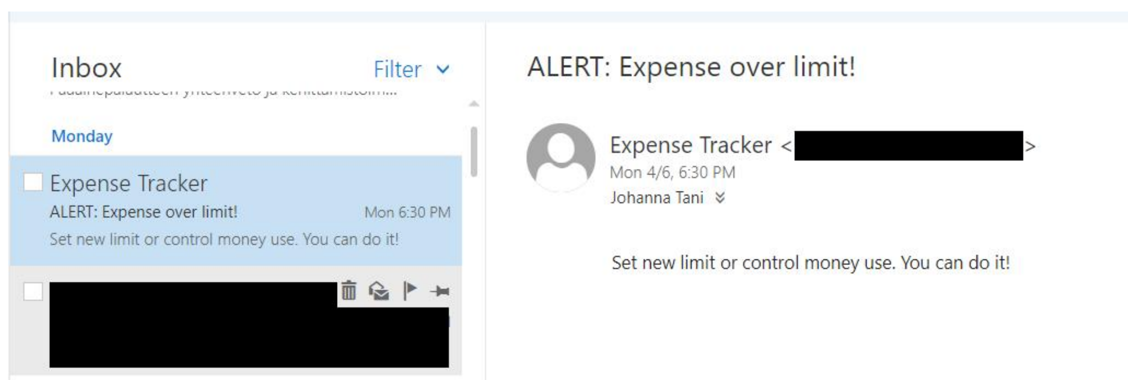
Cloudant-paketin tietokannasta haku, joka haki kaikki kuluvan kuukauden kustannukset. Kolmas toiminto, sum-monthly-expenses, laski kuukauden kustannukset yhteen sekä formatoi vastauksen tietokannasta haku muotoon. Neljännen toiminnon kuuluu hakea tietokannasta kustannusraja kyseiselle kuukaudelle. Tästä muodostui ongelma, sillä valmista tietokannan haku toimintoa ei voinut käyttää ketjussa, sillä kuukauden kustannusten summa piti välittää toiminnossa eteenpäin, jotta sitä ja kustannusrajaa voitiin verrata. Kuitenkin oman toiminnon teko tarkoitti sitä, että tietokannan käyttäjätunnukset jouduttiin välittämään erikseen, sillä pakettisidontaa ei ollut. Muuta vaihtoehtoa ei kuitenkaan ollut, joten valittiin oman toiminnon teko. Jouduttiin siis toteuttamaan oma toiminto tietokannasta hakuun, jossa vastaukseen välitettiin myös kutsussa vastaanotettu kustannussumma. Esimerkkikoodissa 4 on tarkemmin kuvattu, miten vastaukseen välitettiin saatu kustannussumma.

```
function queryIndex(cloudantDb, params) {
	return new Promise(function (resolve, reject) {
		cloudantDb.find(params.query, function (error, response) {
			if (!error) {
				response.sum = params.sum;
				resolve(response);
			} else {
				console.log('error', error);
				reject(error);
			}
		});
	});
}
```

Esimerkkikoodi 4. Metodi itse toteutetusta toiminnosta, jossa välitetään kutsussa vastaanotettu kuukausisumma eteenpäin tietokannasta haetun vastauksen mukana.

Ilman pakettisidontaa toimintoon piti asettaa erikseen oletusparametreja, jossa välittyivät tietokannan käyttäjätunnukset. Oletusparametrit välitetään kutsuun, jos kutsun parametreissa ei ole samannimisiä parametreja, jotka ylikirjoittaisivat ne. Oletusparametrien avulla vältetään pieni tietoturvariski, kun käyttäjätunnuksia ei tarvitse kovakoodata suoraan koodiin.

Viimeinen toiminto ketjussa vertaa kustannusrajaa ja kuukauden yhteenlaskettuja kustannuksia, ja lähettää sähköpostihälytyksen annettuun sähköpostiin, mikäli raja on ylittynyt. Sähköpostin lähetystä varten luotiin erillinen Gmail-sähköpostitili. Sähköpostin lähetyksessä tarvittiin tilin käyttäjätunnukset, joten ne asetettiin koodista piiloon, oletusparametreihin.



Kuva 12. Sähköpostista kuvakaappaus. Kulutusrajan ylitys hälytyssähköposti onnistuneesti saapunut vastaanottajalle.

Toiminnallisuus sähköpostin lähetyksestä saatiin valmiiksi, kun viimeinen toiminto lähetti kuvan 12 mukaisen sähköpostin haluttuun sähköpostiosoitteeseen.

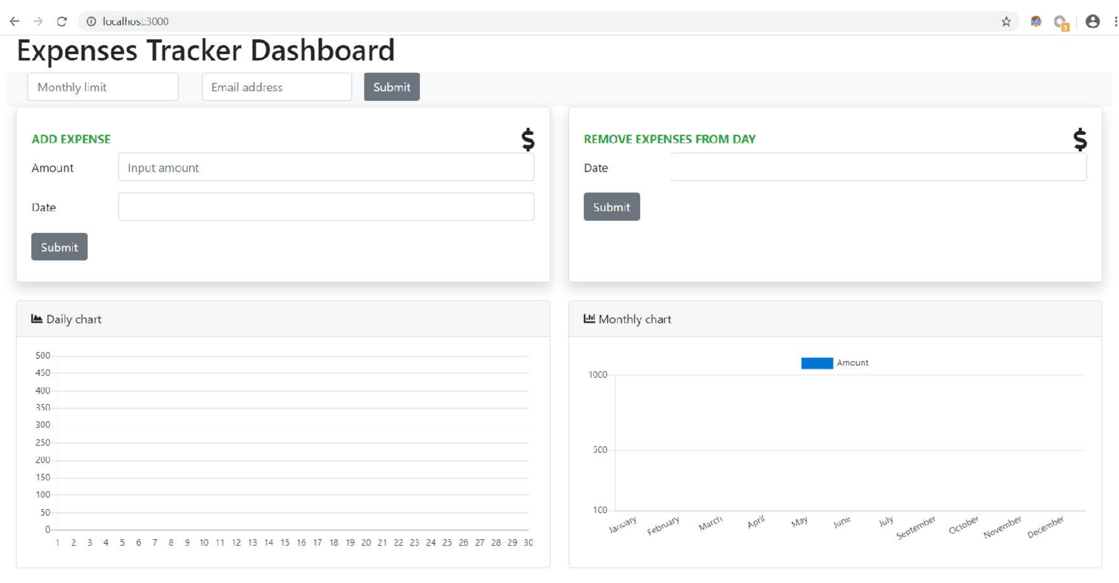
Google estää oletuksena kirjautumiset Gmail-sähköpostiin, mikäli sovellus, josta kirjaudutaan ei vastaa heidän turvallisuusstandardejansa. Nodemailer on Node.js-moduuli, jota käytettiin toiminnossa sähköpostin lähetykseen. Jotta Nodemailer pystyi lähettämään sähköpostin Gmailin kautta, piti ensin sallia Google-tilille kirjautumiset sovelluksista, jotka käyttävät vähemmän turvallisista kirjautumisteknologiaa.

4.5 Front-end-sovellus

Yksinkertainen front-end-sovellus tehtiin, jotta voitiin varmistaa, että toteutettu Serverless backend kommunikoisi sujuvasti front-endin kanssa. Front-end-sovellus toteutettiin käyttämällä Node.js:ää ja Express-sovelluskehystä, sekä käyttöliittymänäkymässä käytettiin Bootstrap-kirjastoa.

Front-end-sovelluksen toteutus aloitettiin luomalla Node.js-palvelininstanssi, joka asetettiin kuuntelemaan porttia 3000. Palvelininstanssin luonnin yhteydessä sovellukselle ladataan staattiset html- ja css-tiedostot. Sovelluksen package.json-tiedostoon määritellään sovelluksen ominaisuudet kuten nimi, versio, riippuvuudet sekä mahdolliset itsemääritellyt skriptit. Jotta sovelluksen käynnistys olisi nopeaa, määritellään skripti package.json-tiedostoon, jolla käynnistetään palvelininstanssi.

Käyttöliittymän kautta haluttiin pystyä kutsumaan kaikkia backend-toiminnallisuuksia. Kuvan 13 mukaisesti, käyttöliittymälle tarvittiin siis kustannuksen lisäys- ja poistoelementit sekä kustannusrajan lisäyselementti. Kuukausittaiset ja päivittäiset kustannukset haluttiin visualisoida diagrammien avulla. Diagrammeihin käytettiin Chart.js:ää, joka on HTML5:n pohjautuva avoimen lähdekoodin kirjasto, jonka avulla saa helposti erilaisia diagrammeja kehitettyä. Aloitettiin luomalla pylväsdiagrammi kuukausittaisille kustannuksille ja viivadiagrammi päivittäisille kustannuksille. Tiedot diagrammeihin haettiin sivun latauksen yhteydessä, sekä tiedot päivitettiin aina lisäyksen ja poiston yhteydessä.



Kuva 13. Käyttöliittymän ulkonäkö ennen backend-kutsujen toteutusta.

Kutsut backend:iin tehtiin käyttäen JavaScript-kirjastoa jQuery, joka helpottaa Ajax-kutsujen tekoa. Ajax on lyhenne sanoista Asynchronous JavaScript And XML ja kuvaa nykyään tekniikkaa, jossa verkkosivuilla JavaScript:illä asynkronisesti tehtävistä http-pyynnöistä palautetaan JSON-muotoinen vastaus. Ennen JSON:ia vastaus oli XML-muodossa, josta nimi on peräisin. Asetettiin jokainen Ajax-kutsu osaksi objektia, jonka avulla kutsuja oli helppo hallinnoida muualla koodissa. Esimerkkikoodi 5:ssa on havainnollistettu objektin sisällä oleva kustannuksen lisäys Ajax-kutsu.

```
const expenses = {
      …
      add(year, month, day, expense) {
            return $.ajax({
                  type: 'POST',
                  url: `${apiUrl}/${entries}`,
                  contentType: 'application/json; charset=utf-8',
                  data: JSON.stringify({
                        year,
```

```
                    month,
                    day,
                    expense
                }),
                dataType: 'json',
            });
        },
        …
{
```

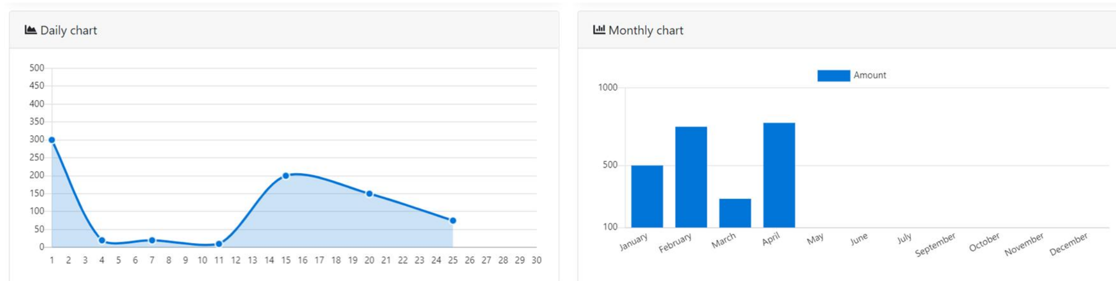
Esimerkkikoodi 5. Expenses-objekti, jonka jokainen sisärakenne palauttaa Ajax-kutsun. Esimerkkiin otettu vain kustannuksen lisäyskutsu.

Objektissa olevat Ajax-kutsut selkeyttävät ja vähentävät duplikaattikoodia. Kun sovelluksessa kutsutaan asynkronisesti backendiä, tarvitaan uudelleenkutsumenettelyä. Uudelleenkutsu (”callback”) on yksinkertaisuudessaan funktio, joka toteutetaan, kun toinen ylemmän luokan funktio on valmis. Uudelleenkutsufunktio välitetään ylemmän luokan funktiolle parametrina. Eli kun Ajax-kutsu tehdään, halutaan tapa käsitellä onnistunut ja epäonnistunut kutsu. JQuery:ssä funktiot done ja fail toimivat halutusti, joten kun pyyntö tehdään, voidaan vastaukset käsitellä done- ja fail-funktioiden avulla, kuten esimerkkikoodi 6:ssa on kuvattu.

```
expenses.add(dateStr[2], dateStr[1], dateStr[0], expense)
      .done(function(result) {
            alert("Expense added.");
            loadYearChart();
            loadMonthChart();
      })
      .fail(function(error) {
            alert("Oops! Something went wrong. Expense was not added.");
            console.log(error);
      });
```

Esimerkkikoodi 6. Kustannuksen lisäyskutsu, jossa käsitellään onnistunut pyyntö lataamalla kuukausittaiset ja päivittäiset kustannukset uudestaan ja näyttämällä hälytys käyttöliittymällä onnistuneella viestillä. Kutsun epäonnistuessa näytetään hälytysviesti ja lokitetaan virheilmoitus konsoliin.

API:n verkko-osoite sekä polut määritellään erillisessä JSON-tiedostossa, josta sovellus käy ne hakemassa. Niin kuin API:n toteutusosuudessa kerrottiin, sen verkko-osoite muuttuu joka kerta, kun API:n käynnistää, joten erillinen tiedosto on helpompi ylläpitää.



Kuva 14. Käyttöliittymän diagrammit sisältävät dataa, kun kutsut toimivat.

Kun kutsut olivat saatu toimimaan API:n kautta, voitiin varmistaa kuvan 14 avulla, että kustannukset haetaan tietokannasta ja ne asettuvat oikein pylväs- ja viivadiagrammeihin.

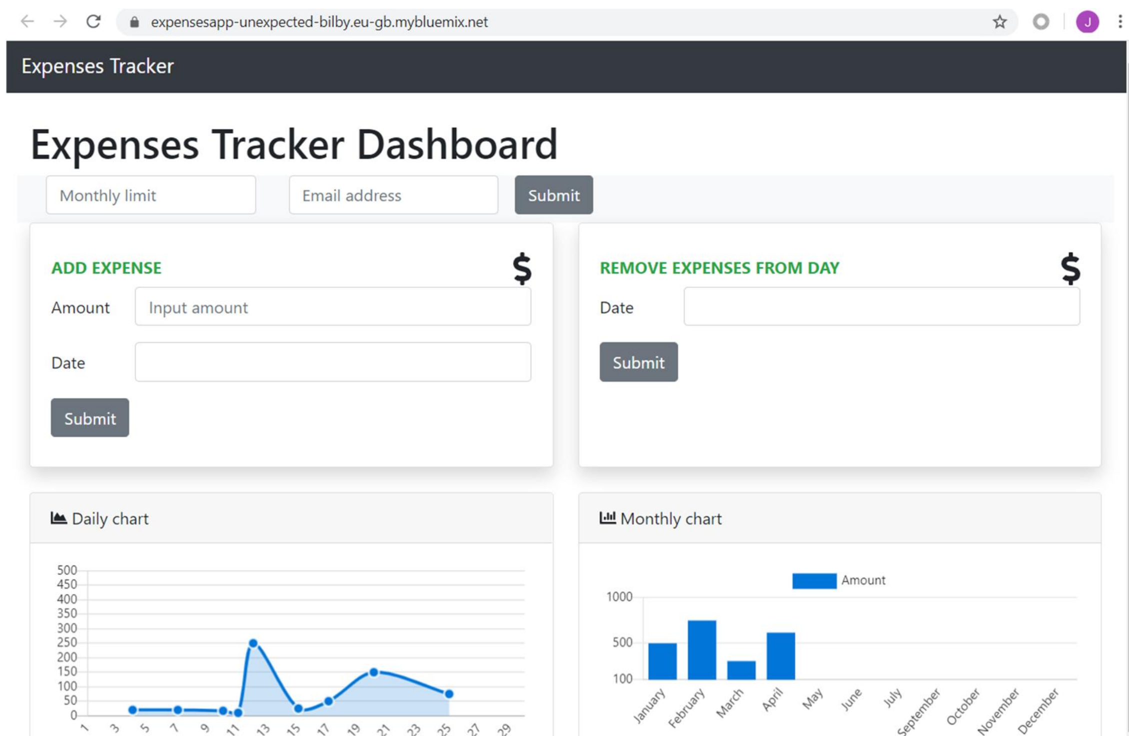
Valmis front-end-sovellus otettiin käyttöön IBM Cloud Foundry -palvelun avulla. Cloud Foundry on PaaS-mallin palvelu, joka tarjoaa alustan, johon front-end-sovellus voidaan asentaa. Sovellukseen määriteltiin manifest.yml-tiedosto, joka sisältää sovelluksen nimen, mahdollisen verkko-osoitteen ja muistin määrän. Tiedoston avulla Cloud Foundry asentaa sovelluksen oikein.

Windows PowerShell

```
PS C:\Users\johanna.tani\oppari\expenses\expenses_app> ibmcloud cf apps
Invoking 'cf apps'...

Getting apps in org                    / space dev as                    ...
OK

name           requested state   instances   memory   disk   urls
expenses_app   started           1/1         128M     1G     expensesapp-unexpected-bilby.eu-gb.mybluemix.net
PS C:\Users\johanna.tani\oppari\expenses\expenses_app>
```

Kuva 15. Komentoliittymän kautta haettu kaikki Cloud Foundry:yn asennetut sovellukset. Tässä tapauksessa front-end-sovellus nimellä expenses_app.

Kun front-end-sovellus on asennettu Cloud Foundryyn, voi komentoliittymän kautta varmistaa, että sovellus on ajossa. Kuvasta 15 näkee Cloud Foundry -alustalle asennetun front-end-sovelluksen tietoja, kuten asetetun muistin ja instanssien määrän sekä sovellukselle generoidun verkko-osoitteen.



Kuva 16. Front-end-sovellus Cloud Foundry-palvelun isännöimänä verkko-osoitteessa expensesapp-unexpected-bilby.eu-gb.mybluemix.net.

Navigoimalla komentoliittymältä nähtyyn verkko-osoitteeseen voidaan kuvan 16 mukaisesti todeta sovelluksen olevan toiminnassa. Näin front-end-sovellusta ei tarvitse käyttää lokaalisti, vaan se on julkisesti käytettävissä Cloud Foundry -palvelun isännöimänä.

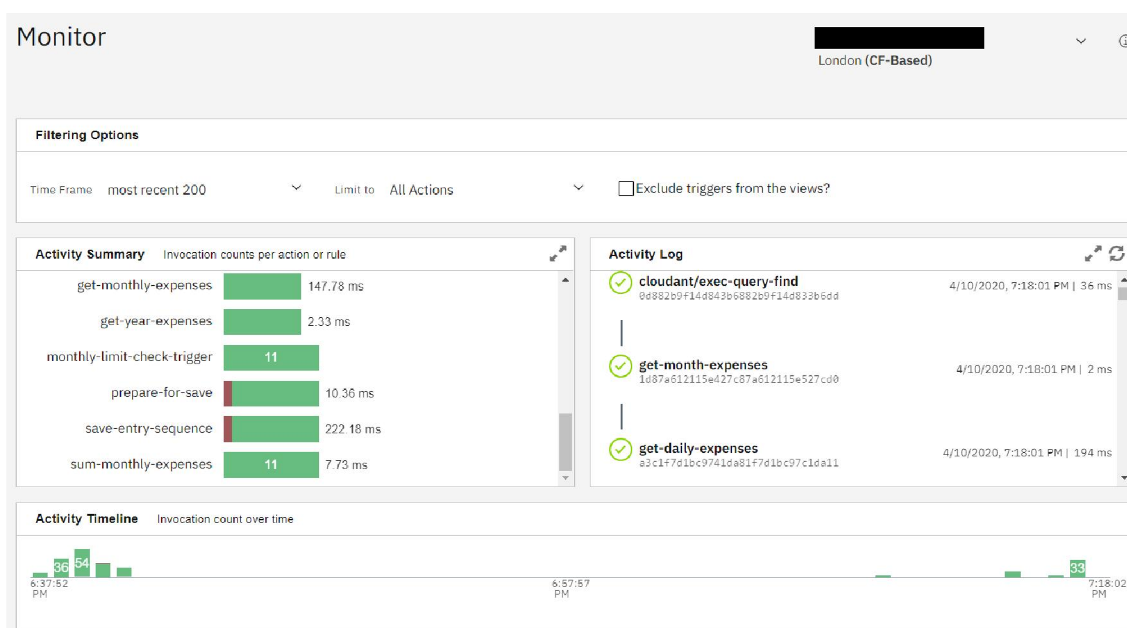
4.6 Monitorointi ja testaus

Virheiden selvitys sekä monitorointi ovat tärkeitä osia sovelluskehityksessä ja sovelluksen ylläpidossa. Tässä osuudessa tutkitaan, miten virheenselvitys hoidetaan FaaS-toimintojen osalta sekä miten toteutettua Serverless-sovellusta voidaan monitoroida.

4.6.1 Monitorointi

IBM Cloud kerää oletusarvoisesti metriikkaa ja lokeja, joita voidaan käyttää pilvialustan ja sovelluksien monitoroinnissa. Tarjolla on myös monitorointiin ja lokien analysointiin palvelut, jotka tarjoavat laajempia monitorointi- ja analysointityökaluja.

Ilmaisella Lite-tilillä ei voi ottaa käyttöön erillisiä monitorointi- tai lokien analysointipalveluja, vaan pitää käyttää valmiiksi integroitua monitorointikapasiteettia, minkä IBM Cloud tarjoaa. IBM Cloud Functions käyttöliittymältä voi siirtyä Monitor-sivulle, jossa on kerättynä viimeisimmistä 200 kutsutusta toiminnosta tietoa. Kuvassa 17 näkyy Monitor-sivu. Sivulla on kolme eri osaa: Ensimmäisessä on toimintokohtaisesti ilmoitettu ajettujen toimintojen lukumäärä, onko toiminto ajettu onnistuneesti, sekä keskiarvo toiminnon kestolle. Toisessa kohdassa on kutsutut toiminnot aikajärjestyksessä, joihin on listattu niiden aktivaatiotunnukset, joita painamalla pääsee katsomaan ajetun toiminnon tiedot ja lokin. Viimeinen kohta on yksinkertainen aikajana, josta näkee, kuinka monta toimintoa on ajettu tiettynä ajankohtana.



Kuva 17. IBM Cloud Functions -käyttöliittymän Monitor-sivu, johon on koottu tietoa suoritetuista toiminnoista.

API:lle on oma yksinkertainen metriikan analysointiosuus, joka löytyy APIs-sivulta käyttöliittymästä. Sivulta näkee vain viimeisen tunnin ajalta analytiikkaa vastauksien määrästä sekä keskiarvoisesta vastausajasta. Sivulla oleva viivadiagrammi kuvamaan API-kutsujen määrää minuutin tarkkuudella. Viimeisenä sivulta löytyy loki API:n vastauksista. Sieltä voi yksitellen tutkia vastauksen tietoja, kuten vastausaikaa ja statuskoodia. Jotta lokiin tallennettaisiin kutsun ja vastauksen sisältö ja ylätunnisteet, kutsussa pitäisi käyttää ylätunnisteena avain-arvoparia ”X-Debug-Mode=true”.

Koska kaikki kerätty data on suhteellisen reaaliaikaista, eikä sitä tallenneta mihinkään, on kattava monitorointi lähes mahdotonta ilmaisella Lite-tunnuksella. Mikäli haluaisi vaihtaa maksullisiin palveluihin, lokien analyysipalvelulla LogDNA:lla kykenisi esimerkiksi lähettämään hälytyksiä tiettyjen lokitapahtumien perusteella ja arkistoimaan tai siirtämään lokitietoja.

4.6.2 Testaus

Virheiden selvitys koodista (”debugging”) on olennainen osa sovelluskehitystä. Kun toiminnot ajetaan eristetyissä konteissa, ajoympäristöön ei päästä käsiksi. IBM Cloud ei tarjoa erillistä palvelua virheenselvitykseen, joten mitään virallista virheenselvitystyökalua ei ole. OpenWhiskistä löytyy käytöstä poistunut OpenWhisk Debugger-työkalu, joten senkään käyttö ei kannata.

Kuten aiemmin on mainittu IBM Cloud Functions käyttää Docker-kontteja ajamaan FaaS-funktionsa. Koska Cloud Functions on rakennettu OpenWhiskin päälle, joka on avoimen lähdekoodin alusta, ajoympäristön määrittelytiedostot (”Docker images”) ovat myös julkisia. Tämä tarkoittaa, että ajoympäristö voidaan simuloida luomalla kontti-instanssi OpenWhiskin julkisen määrittelytiedoston avulla. Simuloituun ympäristöön voidaan aktivoida Node.js:n virheenselvitystyökalu Inspector, jonka jälkeen toimintoja kykenee ajamaan rivi riviltä. Chrome Dev Tools:in avulla asetetaan pysähdyspisteet JavaScript-tiedostoon runner.js, joka ajaa kontti-instanssin toiminnot.

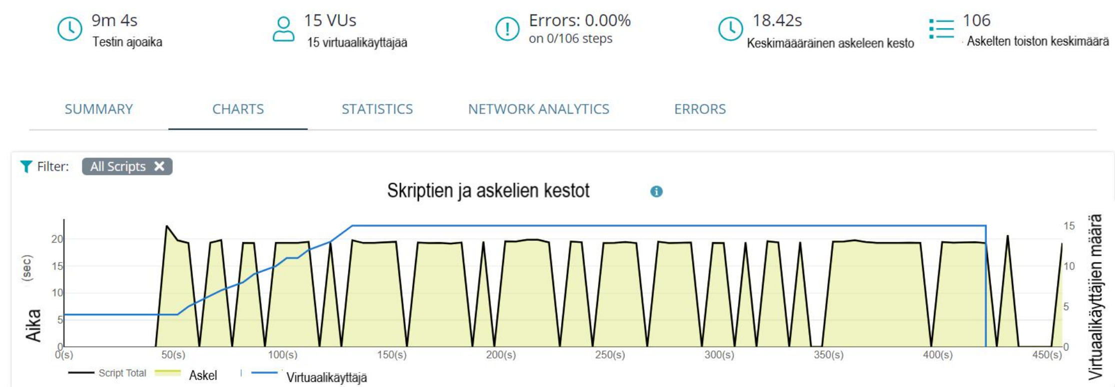
Jotta kutsuja voidaan simuloida, pitää haluttu toiminto importoida ajoympäristöön. Importointi tapahtuu kutsumalla kontissa olevaa päätepistettä init. Kutsuun määritetään pääfunktion nimi ja lähdekoodi. Asetuksen jälkeen toiminto voitiin ajaa kutsumalla päätepistettä run vaadittavien parametrien kanssa. Kun kutsu lähtee, pysäytetään ajo asetettuun pysähdyspisteeseen, jonka jälkeen virheenselvitys onnistuu.

Toteutettu tapa ei ole virallinen, joten tapa simuloida ajoympäristöä voi muuttua. Jokaiselle testattavalle toiminnolle on myös luotava oma kontti, joten ketjutettuja toimintoja ei ole mahdollista testata putkeen.

Suorituskyky- ja kuormitustestauksella varmistetaan sovelluksen toiminta, kun kapasiteettia käytetään enemmän. Serverless-sovelluksen kuuluu skaalautua kuormituksen

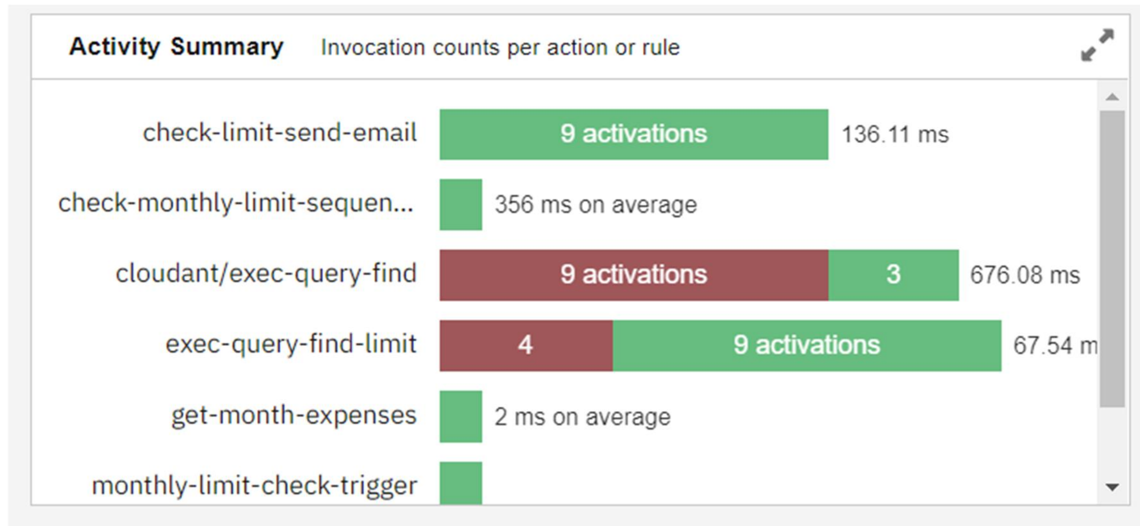
mukaan, joten yksinkertaisella kuormitustestillä voitiin tutkia asiaa. Kuormitustestiin käytettiin LoadNinja-suorituskykytyökalua, joka avulla voi nauhoittaa suoraan käyttöliittymältä testejä ja ajaa ne helposti sen jälkeen ilman suurempia määrittelyjä. Yksinkertainen kuormitustesti tehtiin kustannuksen lisäykselle käyttöliittymältä. Määriteltiin testiin 15 virtuaalista käyttäjää, jotka ovat yhtäaikaisesti sivulla ja lisäävät kaikki kustannuksia.

Testin ajon jälkeen dataa analysoimalla huomattiin, että sovellus ei ylikuormittunut testin myötä. Virheitä ei tullut kuormitustestissä lainkaan, ja vastausajat pysyivät tasaisina käyttäjämäärän noususta huolimatta, kuten kuvasta 18 voi nähdä.



Kuva 18. Kuormitustestin tuloksia. Osa tekstistä on suomennettu selkeyttämään kuvaa.

Kuormitustestin tuloksista voidaan todentaa, että käyttäjien määrän lisääntyminen ei vaikuttanut vastausaikaan, joka pysyi tasaisena testien askelien ajan. Kuvassa keltaiset palkit kuvaavat testin askelien suoritusaikaa ja sininen viiva kuvaa käyttäjien määrää. IBM Cloud Functions Monitor -sivua katsomalla huomattiin, että vaikka sovelluksen käyttöliittymälle ei välittynyt toimintaa aiheuttavia virheitä, osa ajettavista toiminnoista oli epäonnistunut.



Kuva 19. Käyttöliittymäkuva Monitor-sivulta, josta näkee metriikkaa ajetuista toiminnoista. Kuvassa tietokannastahaku toiminnot ovat epäonnistuneet.

Kuvasta 19 huomaa, että osa tietokantahauista oli epäonnistunut. Toimintolokeja tutkimalla selvisi, että Lite-tilillä on tietokannassa kapasiteettirajoituksia. Sallittu kapasiteetti on 20 lukuoperaatiota sekunnissa ja 10 kirjoitusoperaatiota sekunnissa, mikä selittää toimintojen epäonnistumisen. Tietokannan kapasiteettirajoitusten takia voidaan kuormitustestauksen myötä todeta, että sovelluksen suorituskyky toimii ainakin rajoituksiin asti.

5 Arviointi

Testisovelluksen toteutuksen jälkeen arvioitiin valmis sovellus. Arvioinnissa käytettiin aikaisemmin tehtyä vaatimus- ja prioriteettimäärittelyä ja tutkittiin sen pohjalta, onnistuttiinko toteutuksessa halutulla tavalla. Jokaiselle vaatimukselle annettiin vaihtoehdoksi täysin onnistunut toteutus, osittain onnistunut toteutus ja epäonnistunut toteutus. Vaihtoehtojen pohjalta tutkittiin erikseen jokaiselle vaatimukselle sopiva kohta.

Prioriteetilla 1 oleva vaatimus saatiin toteutettua, sillä sovelluksen käyttöliittymältä pystyi kutsumaan backend-toimintoja REST API:n kautta, johon oli määritelty halutut API-operaatiot. Vaatimukset prioriteeteilla 2 ja 3 saatiin myös toteutettua, mitä tarkoitti, että koko sovelluksen vaadittu backend-toiminnallisuus saatiin tehtyä vaatimusten mukaisesti. Sovellus kommunikoi tietokannan kanssa, lähetti hälytyksen tarpeen mukaan, sekä toiminnot ajettiin API-kutsujen perusteella, kuten määrittelyssä oli vaadittu. Vaatimus prioriteetilla 4 onnistui vain osittain, sillä kuormituskykytestiä tehdessä huomattiin tietokannalle

asetetut rajoitukset, jotka estivät tietokantakyselyjen määrää. Tämän vuoksi voitiin todeta vain, että tehty API ja siihen liitetyt toiminnot skaalautuivat liikenteen lisäyksen myötä, mutta tietokanta ei, vaikka Cloudant BaaS-palvelun tiedoissa luvattiin skaalautuvuutta. Prioriteetilla 5 oleva vaatimus, jonka mukaan sovellusta pystyi testaamaan, asetettiin osittain onnistuneeksi sen takia, koska toteutuksessa käytetty virheenselvitystapa ei ollut virallinen ja voi mahdollisesti tulevaisuudessa muuttua mahdottomaksi. Vaatimus prioriteetilla 6 on osittain onnistuneen ja epäonnistuneen rajalla, sillä sovelluksen monitorointiin ei voitu toteuttaa mitään omaa toiminnallisuutta. Valmiiksi tarjottu reaaliaikainen monitorointi oli ainoa mahdollinen keino monitoroida sovellusta. Vaatimus asetettiin kuitenkin osittain onnistuneeksi, sillä vaikka omia monitorointihälytyksiä ei kyennyt tekemään, eikä vanhoja lokitietoja ei voinut selata, tarjottu reaaliaikainen monitorointi oli mahdollista.

Taulukko 3. Taulukko kuvaa, miten vaatimusmäärittelyssä asetetut vaatimukset onnistuttiin toteuttamaan.

Vaatimus	**Prioriteetti**	**Toteutus onnistui**	**Toteutus onnistui osittain**	**Toteutus ei onnistunut**
Sovellus käyttää REST API:a, ja vastaa http-kutsuihin: GET/POST/DELETE	1	X		
Sovellus tallentaa ja hakee arvoja tietokannasta.	2	X		
Sovellus reagoi tietokantamuutokseen lähettämällä sähköpostin, mikäli kuukausittainen kulutusraja ylittyy.	3	X		
Sovellus skaalautuu tarpeen mukaan.	4		X	
Sovellusta kykenee testaamaan.	5		X	
Sovellusta kykenee monitoroimaan	6		X	

Taulukkoa 3 tutkimalla voidaan todeta, että kuudesta vaatimuksesta 50 prosenttia onnistuttiin toteuttamaan täysin onnistuneesti ja toiset 50 prosenttia osittain onnistuneesti. Jos annetaan onnistuneesti toteutetuille vaatimuksille prosenttiarvo 100 ja osittain onnistuneille vaatimuksille arvo 50, voidaan karkeasti laskea, että kehitetty sovellus onnistui täyttämään 75 prosenttia halutuista määrityksistä. Onnistuneesti toteutetut vaatimukset

olivat prioriteeteiltaan tärkeimmät ja osittain onnistuneet vaatimukset prioriteettilistan loppupäässä. Prioriteeteiltaan tärkeimmät vaatimukset olivat olennaisia osia sovelluksen toiminnassa, kun taas alemmalla prioriteetilla oli ylläpitoon ja sovelluksen hallintaan tarvittavia osia. Antamalla prioriteeteille painoarvoa, voidaan laskea, että sovellus onnistui noin 86-prosenttisesti. Tulokseen päädyttiin asettamalla jokaiselle prioriteetille oma painoarvonsa. Prioriteetti 1 sain arvon 6, prioriteetti 2 sain arvon 5 ja niin edelleen. Summattiin painoarvot yhteen niin, että osittain onnistuneiden vaatimusten painoarvot kerrottiin 0,5:llä ja onnistuneiden yhdellä. Painoarvojen summaksi tuli 18. Maksimipainoarvo oli 21, joten jaettiin saatu summa maksimiarvolla ja saatiin tulokseksi 0,8571, pyöristettynä 0,86.

Käytetty IBM Cloud -kehitysalusta oli tärkeä osa sovelluksen kehityksessä, sillä sen avulla toteutettiin Serverless-arkkitehtuurin mukainen backend. Tarkastellaan kehitysalustan toimivuutta muutaman kysymyksen avulla.

- Minkälainen dokumentaatio oli?
- Oliko kehitysalusta helppokäyttöinen?
- Rajoittiko kehitysalusta toteutusta?
- Sopisiko vaativampaan käyttöön?

Kehitysalustan dokumentaatio oli laajaa ja helposti löydettävää. Sitä myös ylläpidettiin aktiivisesti, mikä teki kehitysalustan käyttöönotosta helppoa. Alusta oli ilman dokumentaatiotakin todella helppokäyttöinen, eikä vaatinut pitkää tutustumista tai opettelua. Selainkäyttöliittymältä oli hyvin helppoa esimerkiksi luoda Cloud Functionsin kautta toimintoja tai hallinnoida Cloudant-tietokantaa. Komentoliittymän käyttö oli hieman vaativampaa ilman visuaalista puolta, mutta oli kaiken kaikkiaan nopeasti opittava. Testisovelluksen toteutuksessa käytetty ilmainen Lite-käyttäjätili asetti palveluiden kapasiteeteille rajoituksia, mikä rajoitti sovelluksen toiminnallisuutta. Rajoitteiden takia sovelluksen tietokanta ei skaalautunut halutulla tavalla, monitorointihälytyksiä tai lokien tallennuksia ei voinut tehdä, eikä API-palvelulle saanut luotua omaa verkko-osoitetta. Ottamalla käyttöön maksulliset palvelut, mainitut rajoitteet on mahdollista mitätöidä. FaaS-toimintojen virheenselvitys kooditasolla ei kuitenkaan ole mahdollista edes maksullisena, joten se on kehitysalustan rajoitteena käyttäjätili- ja palvelutyypistä riippumatta. Testisovelluksen rajoitusten takia on mahdotonta arvioida, sopiiko kehitysalusta vaativampaan käyttöön, sillä kunnollisia suorituskyky- tai kuormitustestejä ei voitu tehdä.

IBM Cloud -kehitysalustan ilmainen käyttäjätili ei ole toimiva web-sovelluskehityksessä rajoitteidensa takia. Maksullisia palveluita käyttämällä IBM Cloud on helppokäyttöinen kehitysalusta, joka sopii ainakin yksinkertaisten Serverless-sovelluksien kehitykseen.

6 Yhteenveto

Insinöörityön tavoitteena oli perehtyä Serverless-arkkitehtuuriin sekä tutustua Serverless-, FaaS- ja BaaS-käsitteisiin tarkemmin. Lisäksi tavoitteena oli arvioida Serverless-arkkitehtuurin toimivuutta web-sovelluskehityksessä toteuttamalla vaatimusmäärittelyn mukainen testiverkkosovellus arkkitehtuuria noudattamalla. Toteutuksen avulla arvioitiin myös toteutuksessa käytettyä kehitysalustaa IBM Cloudia.

Insinöörityön teoriaosuudessa käytiin läpi, mitä Serverless-arkkitehtuuri tarkoittaa ja mitä sen sisältämät pilvipalvelumallikäsitteet FaaS ja BaaS tarkoittavat sekä mikä niiden suhde on toisiinsa. Työssä tarkasteltiin Serverless:in hyötyjä ja haasteita ja suhdetta muihin pilvipalvelumalleihin. Työn toteutusosassa määriteltiin testisovellus, jolle annettiin kuusi toiminnallista vaatimusta, jotka asetettiin prioriteettijärjestykseen. IBM Cloud -kehitysalustaa käyttäen toteutettiin kustannusten seuranta -sovellus noudattamalla Serverless-arkkitehtuurin periaatteita. FaaS-toimintojen kehitykseen käytettiin IBM Cloud Functions PaaS -palvelua ja tietokantana Cloudant BaaS -palvelua. Sovelluksen toteutuksen jälkeen arvioitiin, vastaako toteutettu sovellus vaatimusmäärittelyä. Arvioinnin perusteella kolme vaatimusta saatiin toteutettua täysin onnistuneesti ja kolme vaatimusta osittain onnistuneesti. Arvioinnissa annettiin vaatimuksille toteutusarvot, joiden perusteella laskettiin prosenttimääräisesti, että sovellus saatiin toteutettua 75-prosenttisesti. Ottamalla huomioon prioriteettipainotuksen sovellus saatiin toteutettua 86-prosenttisesti. Arvioinnissa tarkasteltiin myös käytettyä kehitysalustaa arvioimalla sen toimintaa neljän eri kysymyksen avulla.

Työn avulla saatiin selvitettyä, että Serverless on kevyt tapa kehittää verkkosovelluksia, joiden backend-toiminnallisuus ei vaadi monimutkaisia ja aikaa vieviä operaatioita. Erilaiset verkkokaupat ovat hyviä esimerkkejä sovelluksista, joissa Serverless-arkkitehtuuri olisi helppo toteuttaa. Pelisovellukset tai sovellukset, jossa käsitellään Big Dataa, eivät ole parhaita kehittää Serverless-arkkitehtuurin mukaan vastausviiveen ja lyhytaikaisuutensa takia. Kehitysalustalle asetettujen arviointikysymysten perusteella voitiin todeta,

että maksamalla palveluiden käytöstä, IBM Cloud ja sen tarjoamat palvelut sopivat Serverless-verkkosovellusten tekoon.

Työlle asetetut tavoitteet saavutettiin hyvin ja haluttu toteutus saatiin toteutettua vaatimusten mukaisesti suurimmaksi osaksi kokonaan. Toteutettua saatiin toimiva Serverless-arkkitehtuuria noudattava sovellus, jossa käytettiin FaaS-funktioita toteuttamaan haluttu backend sekä Cloudant BaaS -palvelua tietokantapalveluna. Front-end-sovellus asennettiin Cloud Foundry -palveluun ajoon, joten koko sovelluksen palvelinpuolen logiikka oli IBM Cloudin vastuulla. IBM Cloud oli kehitysalustana sopiva, kun toteutettiin pieni testisovellus. Ilmainen Lite-käyttäjätili on selvästi tehty, jotta mahdolliset asiakkaat voisivat tutustua paremmin IBM Cloud -tuotteisiin ilman velvoitteita. Sovelluskehitykseen ei ilmaista käyttäjätiliä voi käyttää, niin kuin työssä on todettu. Serverless-arkkitehtuurin mukaan toteutettu backend onnistuttiin toteuttamaan täysin ilman suuria ongelmia tai kompromisseja. Toteutuksessa huomattiin teoriaosuudessa mainittuja FaaS-funktioiden ominaisuuksia, kuten esimerkiksi käynnistysviivettä toimintojen ajossa.

Insinöörityössä jäi selvittämättä, miten testisovelluksen toteutus olisi onnistunut maksamalla käytetyistä palveluista. Näin ilmaisten palveluiden rajoitteet eivät olisi estäneet sovelluksen toimintaa. Selvittämättä jäi myös, miten muiden yritysten palveluiden integraatio sovellukseen olisi onnistunut IBM:n kehitysalusta kanssa. Insinöörityötä voisi jatkaa selvittämällä, miten eri Serverless-kehitysalustat eroavat toisistaan. Pilvipalveluidentarjoajilla on kaikilla omat kehitysalustansa, joten työtä jatkamalla voisi tutkia, miten sovelluskehitys eroaisi kehitysalustojen välillä. Vertailussa voisi myös tutkia, kuinka palveluntarjoajista riippuvaisia sovelluksista tulisi. Monen kehitysalustan välillä voisi tehdä sovellusmigraation ja tutkia, kuinka hankalaa on vaihtaa palveluntoimittajasta toiseen.

Työtä voidaan hyödyntää, kun halutaan tietoa Serverless-käsitteestä tai Serverless-arkkitehtuurista ja sen sisältämistä pilvipalvelumalleista BaaS ja FaaS. Työn avulla saatiin lisäksi käytännön tietoa Serverless-arkkitehtuurin mukaisesta web-sovelluksen kehityksestä sekä IBM Cloud -kehitysalustasta ja sen käytöstä.

Lähteet

1 What is Serverless Architecture? What are its Pros and Cons? 2018. Verkkoaineisto. <https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9> Hackernoon. Luettu 18.1.2020.

2 Serverless Architecture. Verkkoaineisto. <https://www.twilio.com/docs/glossary/what-is-serverless-architecture> Twilio. Luettu 12.1.2020.

3 Serverless. Verkkoaineisto. <https://aws.amazon.com/serverless/> AWS. Luettu 1.2.2020.

4 Serverless Architectures. 2018. Verkkoaineisto. <https://martinfowler.com/articles/serverless.html> Martin Fowler. Luettu 18.1.2020.

5 What is BaaS? Verkkoaineisto. <https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/> CloudFare. Luettu 12.1.2020.

6 What is Serverless? Verkkoaineisto. <https://serverless-stack.com/chapters/what-is-serverless.html> Serverless Stack. Luettu 18.1.2020.

7 Pros and Cons of Serverless Computing. FaaS comparison: AWS Lambda vs Azure Functions vs Google Function. 2019. Verkkoaineisto. <https://assist-software.net/blog/pros-and-cons-serverless-computing-faas-comparison-aws-lambda-vs-azure-functions-vs-google> Assist. Luettu 19.1.2020

8 When are Cold Starts a Problem? 2019. Verkkoaineisto. <https://www.nuweba.com/blog/when-are-cold-starts-problem> Ido Neeman. Luettu 19.1.2020.

9 BaaS vs FaaS – What's the Difference? 2019. Verkkoaineisto. <https://blog.back4app.com/2019/10/23/baas-vs-faas/> George Batschinski. Luettu 8.2.2020.

10 What is serverless architecture? Pros, cons, and how to get started. 2018. Verkkoaineisto. <https://learntocodewith.me/posts/serverless-architecture/> Robert Fisher. Luettu 19.1.2020.

11 IaaS, CaaS, PaaS, FaaS, SaaS – mitä mikäkin tarkoittaa? 2020. Verkkoaineisto. <https://onrego.fi/julkisen-pilven-palvelumallit-avattuna/> Jussi Vento. Luettu 7.4.2020.

12 AWS Simple Monthly Calculator. 2020. Verkkoaineisto. <https://calculator.s3.amazonaws.com/index.html#r=IAD&key=calc-40B81990-44E5-434C-9DC2-94B273D1B1F2> AWS. Luettu 8.2.2020.

13 Why use Serverless Computing? | Pros and Cons of Serverless. Verkkoaineisto. <https://www.cloudflare.com/learning/serverless/why-use-serverless/> Luettu 19.1.2020.

14 Regions, Availability Zones, and Local Zones. Verkkoaineisto. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html> AWS. Luettu 19.1.2020.

15 Apache OpenWhisk. Verkkoaineisto. <https://openwhisk.apache.org/> Luettu 25.1.2020.

16 What container? Verkkoaineisto. <https://www.docker.com/resources/what-container> Docker. Luettu 25.1.2020.

17 Cloud Functions Terminology. 2020. Verkkoaineisto. < https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-about#about_technology> IBM Cloud. Luettu 3.4.2020.

18 Introduction. Verkkoaineisto. <https://docs.couchdb.org/en/stable/intro/index.html> Apache CouchDB. Luettu 31.3.2020.

19 IBM Cloudant. Verkkoaineisto. <https://www.ibm.com/fi-en/marketplace/database-management> IBM. Luettu 31.3.2020.

20 Comparing Apache CouchDB and IBM Cloudant. 2020. Verkkoaineisto. <https://cloud.ibm.com/docs/Cloudant?topic=cloudant-couchdb-and-cloudant> IBM Cloud. Luettu 31.3.202