SAVONIA

OPINNÄYTETYÖ - AMMATTIKORKEAKOULUTUTKINTO
TEKNIIKAN JA LIIKENTEEN ALA

PALVELININFRASTRUKTUURIN TOTEUTTAMINEN OHJELMALLISESTI AWS-PALVELUITA HYÖDYNTÄEN

TEKIJÄ/T: Joni Pössi

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Tutkinto-ohjelma Tietotekniikan tutkinto-ohjelma	
Työn tekijä(t) Joni Pössi	
Työn nimi Palvelininfrastruktuurin toteuttaminen ohjelmallisesti AWS-palveluita hyödyntäen	
Päiväys 22.5.2021	Sivumäärä/Liitteet 21/0
Toimeksiantaja/Yhteistyökumppani(t) Nursie Health Oy	

Tiivistelmä

Opinnäytetyön tarkoituksena oli syventyä palvelininfrastruktuurin toteuttamiseen ja kehittämiseen ohjelmallisesti käyttämällä Amazonin tarjoamia palveluita ja kehitystyökaluja, jotta nykyisen järjestelmän kehitys sujuisi ketterämmin nyt ja tulevaisuudessa.

Työ alkoi perehtymällä infrastruktuurin ohjelmallisuuden hyötyihin ja toteutustapoihin sekä Amazonin palveluiden luontiin ja niiden toimintaperiaatteisiin. Toteutettu infrastruktuuri koostuu käyttäjäryhmästä, tietokannasta ja API-päätepisteistä, jotka luotiin täysin ohjelmallisesti JavaScript-kielellä käyttäen Serverless Stack- ja Cloud Development Kit -kirjastoja.

Luotujen palveluiden toiminta todettiin React Native -mobiilisovelluksella. Sovellus käyttää käyttäjäryhmää rekisteröinti- ja kirjautumistoimintoihin sekä käyttäjän syöttämien tapahtumien hakuun, lisäämiseen ja muokkaukseen API-päätepisteitä, jotka puolestaan kommunikoivat tietokannan kanssa. Toimintojen toteutuksessa hyödynnettiin Amplify-kirjastoa.

Lopputuloksena syntyi toimiva infrastruktuuri ja mobiilisovellus toiminnallisuuksineen. Projektin kehitys opetti paljon Amazonin palveluista ja sen konsepteista ja antoi hyvän pohjan vastaavien järjestelmien kehitykselle tulevaisuudessa.

Avainsanat

taustajärjestelmä, AWS, infrastruktuuri ohjelmallisesti, pilvipalvelut, React Native, JavaScript

Field of Study Technology, Communication and Transport	
Degree Programme Degree Programme in Information Technology	
Author(s) Joni Pössi	
Title of Thesis Setting Up Server Infrastructure Programmatically Utilizing Amazon AWS	
Date May 22, 2021	Pages/Appendices 21/0
Client Organisation /Partners Nursie Health Oy	

Abstract

The purpose of this thesis was to learn and research about deploying and developing server infrastructure programmatically using services and tools provided by Amazon so that the current system could be developed more efficiently now and in the future.

The work started by getting familiar with the benefits and implementation approaches of deploying infrastructure as code, as well as different ways to create Amazon services and how they work. The created infrastructure consists of a user pool, database and multiple API endpoints, which were created entirely in JavaScript using Serverless Stack and Cloud Development Kit frameworks.

The operation of the created services were verified by utilizing them in a mobile application created with the React Native framework. The application uses the user pool for authentication operations such as registration and logging in. User submitted events are fetched, created and modified by calling API endpoints which in turn communicate with the database. The functions were implemented by utilizing the Amplify framework.

The result was a working server infrastructure and a mobile application to use it with. Developing the project taught a lot about Amazon services and their development tools, which gives a basis for developing similar projects in the future.

Keywords
back end, AWS, infrastructure as code, cloud, React Native, JavaScript

SISÄLTÖ

KUVALUETTELO

1 JOHDANTO

Pilvipalvelut ovat viime vuosikymmenen aikana kasvaneet suosioon niiden pienempien kustannusten, skaalautuvuuden ja ylläpidettävyyden ansiosta. Pilvipalvelut ei itsestään kuitenkaan ratkaise kaikkia fyysisille palvelimille ominaisia ongelmia ja haasteita, joista muutamia ovat käyttöönotto, konfiguraatio ja purku, joten niille oli kehitettävä jokin ratkaisu. Yksi näistä on malli, jossa palvelininfrastruktuuri määritetään koodina koostuen avain-arvo-pareista eri muotoisissa tekstitiedostoissa konfigurointityökalujen sijaan. Luonnollinen jatke tälle mallille on käyttää ohjelmointikieltä infrastruktuurin määrittämiseen, johon tässä opinnäytetyössä tutustutaan käyttämällä Amazonin tarjoamia palveluita.

Työn tilaajana toimi Nursie Health Oy, joka on tätä kirjoittaessa myös nykyinen työnantajani. Idea aiheeseen lähti tarpeesta ja halusta oppia lisää töissä käytetyistä tekniikoista, sillä vaikka uusien ominaisuuksien lisääminen ja muutosten tekeminen luonnistui, en kuitenkaan tiennyt tarkemmin sen toimintaperiaatteista tai kuinka vastaavanlaista järjestelmää lähdettäisiin rakentamaan. Nykyisen infrastruktuurin kehitys on ostettu kolmannen osapuolen yritykseltä, mutta kesän aikana sen kehitys siirtyy vähitellen meidän alaisuuteemme, joten kaikki tietotaito järjestelmästä tulee tarpeeseen.

Opinnäytetyössä tullaan tutustumaan infrastruktuuri koodina -mallin hyötyihin ja toimintatapoihin sekä käytettyjen työkalujen ja tekniikoiden toimintaprosesseihin, jonka jälkeen toteutetaan taustajärjestelmä käyttämällä kyseisiä tekniikoita. Taustajärjestelmän palveluita hyödyntävien toiminnallisuuksien toteutusta tullaan tarkastelemaan React Native -mobiilisovelluksen näkökulmasta. Itse sovelluksen kehitykseen ei tulla perehtymään, sillä opinnäytetyön pääpaino on infrastruktuurissa.

2 TEORIA

2.1 Infrastruktuuri koodina

Mitä tarkoitetaan termillä ”infrastruktuuri koodina” (engl. infrastructure as code) ja mitä ongelmaa sillä pyritään ratkaisemaan? Perinteisesti taustajärjestelmät, kuten tietokannat ja verkon tallennustilat on asennettu, konfiguroitu ja otettu käyttöön manuaalisesti. Yleensä tämä käsittää fyysisten laitteiden, käyttöjärjestelmien ja ohjelmistojen asentamista ja muokkaamista, joka vaatii paljon kokenutta henkilökuntaa, aikaa ja rahaa, jotta koko prosessia voidaan hallita tehokkaasti. Tämä on tehtävä aina uudestaan, kun vanhat laitteet päivitetään uusiin tai olemassa olevaa infrastruktuuria laajennetaan uusilla laitteilla. (Schults, 2019.)

Infrastruktuuri koodina taas on tapa, jolla koko taustalla toimivan järjestelmän resurssit ja ominaisuudet määritellään joko terminaalissa ajettavilla komennoilla, ohjelmointikielellä tai jossain ihmisen luettavassa tiedostomuodossa, kuten JSON (JavaScript Object Notation) tai YAML (YAML Ain’t Markup Language). Kun määritykset on tehty, ne ajetaan asiaan kuuluvan työkalun läpi, joka rakentaa resurssit juuri halutulla tavalla. Usein kohteena on jokin pilvipalvelin tai -palvelu. (Schults, 2019.)

2.1.1 Lähestymistavat

Infrastruktuurin määrittämistä voi lähestyä kahdella tavalla, deklaratiivisesti ja imperatiivisesti. Deklaratiivisessa tavassa resurssit määritellään ohjelmointikielellä tai sopivassa tekstitiedostossa, kuten JSON- ja YAML-tiedostoissa. Tässä menetelmässä käytetyt työkalut automatisoivat resurssien viemistä palvelimelle sekä helpottavat muutosten tekemistä ja infrastruktuurin purkamista, mutta vaatii ylläpitäjältä paljon osaamista käytetystä kielestä ja teknologioista. (IBM Cloud Education, 2019.)

Imperatiivisessa tavassa taas resurssit luodaan ajamalla CLI-käyttöliittymässä (Command Line Interface) komentoja yksikerrallaan oikeassa järjestyksessä, yleensä hyödyntämällä skriptausta. Tämä menetelmä on ylläpitäjille helpompi ymmärtää ja soveltaa aiemmin tehtyjä skriptejä, mutta infrastruktuurin kasvaessa työläänpää hallita kuin deklaratiivinen menetelmä. (IBM Cloud Education, 2019.)

Edellä mainittujen asioiden lisäksi infrastruktuuria automatisoidessa on tehtävä päätös sen muuttuvuudesta. Muuttuvassa infrastruktuurissa käyttöönoton jälkeen voidaan tehdä suunnittelemattomia muutoksia sen konfiguraatioon esimerkiksi korjaamaan tietoturvariski tai mukauttamaan ympäristöä sopivammaksi sitä käyttävää sovellusta varten. Tämä kuitenkin voi johtaa dokumentoimattomiin muutoksiin eikä täysin samanlaista ympäristöä voida enää toistaa, jolloin menetetään ne hyödyt, jotka koodina määritetty infrastruktuuri tuo. Muuttumattomassa infrastruktuurissa muutokset tehdään aina koodiin tai määritystiedostoihin, jonka jälkeen uusi konfiguraatio viedään palvelimelle ja otetaan käyttöön. Näin toimimalla ympäristön tila on aina tiedossa ja se voidaan toistaa. (IBM Cloud Education, 2019.)

2.1.2 Hyödyt

Yleisesti pilvessä toimivien palvelinten ja palvelujen vahvuuksia ovat nopeus, skaalautuvuus ja kustannustehokkuus. Nopeudella tarkoitetaan sitä, kuinka nopeasti infrastruktuuri saadaan pystytettyä

ja purettua. Uusia laitteita ei tarvitse tilata ja aikaa ei kulu niiden toimitukseen. Aikaa ei myöskään kulu manuaaliseen työhön kuten fyysisten laitteiden asennukseen, sekä palvelun vaatimien käyttöjärjestelmien ja sovellusten asennukseen ja konfigurointiin. Sama pätee myös skaalautuvuuteen, sillä suorituskapasiteettia on helpompi lisätä kysynnän mukaan. Kaikki tämä johtaa lopuksi pienempiin kustannukseen; kalliisiin laitteisiin ja konesaleihin ei ole tarvetta sijoittaa, mikä on pienemmille ja uusille yrityksille houkuttelevaa, ja laskentatehosta maksetaan vain kulutuksen mukaan. (Schults, 2019.)

Kun infrastruktuuri on määritelty koodina, saadaan pilvipalveluista yhä enemmän hyötyä irti. Uudet palvelut voidaan ottaa nopeasti käyttöön ajamalla koodi käyttöönottotyökalun läpi. Luonteensa ansiosta siitä voidaan tehdä modulaarinen, jolloin infrastruktuurin eri osia voidaan hyödyntää toisissa projekteissa (Red Hat, julkaisuaika tuntematon). Tuotanto- ja kehitysympäristöt on helppo pitää erillään ja projektiryhmän jäsenille saadaan myös pystytettyä muutamalla komennolla omat ympäristönsä, jolloin ei tarvitse varoa toisten työn häiritsemistä.

Koska kyseessä on ohjelmakoodi, on hyvien tapojen mukaista käyttää silloin hyödyksi versionhallintatyökalua, kuten Git:iä. Versiohallinnan avulla pysytään perillä siitä, mitä muutoksia infrastruktuuriin on tehty, kuka sen teki ja miksi. Koodi myös kertoo selkeästi sen, millainen infrastruktuuri kokonaisuudessaan on. Toisin sanoen koodi itsessään toimii dokumentaationa infrastruktuurista, jolloin ei välttämättä edes tarvita erillistä dokumentaatiota. Tämä voi kuulostaa oudolta käytännöltä, mutta näin toimimalla koodista muodostuu se yksi ainoa totuus koko infrastruktuurin tilasta, eikä koodin ja dokumentaation välille pääse syntymään epäyhtenäisyyksiä. (Schults, 2019.)

2.2 Amazon Web Services

Amazon Web Services (AWS), on Amazonin tarjoama pilvipalvelualusta, joka sisältää lukuisia virtuaalikone-, tietokanta-, koneoppimis- ja analytiikkapalveluita sekä paljon muuta. Monet AWS:n palvelut ovat käytettävissä ilmaiseksi tietyin rajoituksin, jotka kuitenkin sopivat tämän mittakaavan työhön oikein hyvin. Tässä opinnäytetyössä käytettiin pääasiassa Amplify-, Lambda-, DynamoDB-, Cognito- ja API Gateway -palveluita, joiden toteutus käydään tarkemmin luvuissa 3 ja 4.

2.2.1 AWS Konsoli

AWS konsoli on perinteinen web-selaimen kautta käytettävä käyttöliittymä, jolla palveluita voidaan hallita ja luoda. Kaikki työ on täysin manuaalista, ja ei siksi ole se paras tapa hallita useaa ympäristöä samasta infrastruktuurista, sillä täysin samanlaista ympäristöä ei voida taata ja se on työlästä toteuttaa.

Pääkäyttäjätunnuksen sijaan konsolin käyttöä varten tehdään hyvien tapojen mukaisesti jokaiselle projektiryhmän jäsenille oma IAM-tunnus. IAM on ominaisuus, jolla voidaan luoda käyttäjiä ja ryhmiä ja asettaa niille rooleja, joilla voidaan esimerkiksi rajata käyttöoikeuksia vain tiettyihin resursseihin (Amazon Web Services, 2021). Kun infrastruktuuri toteutetaan koodina, käytetään konsolia suurimmaksi osaksi työntekijöiden IAM-tunnuksien hallintaan, luotujen resurssien ja loki- ja tietokantatapahtumien tarkasteluun.

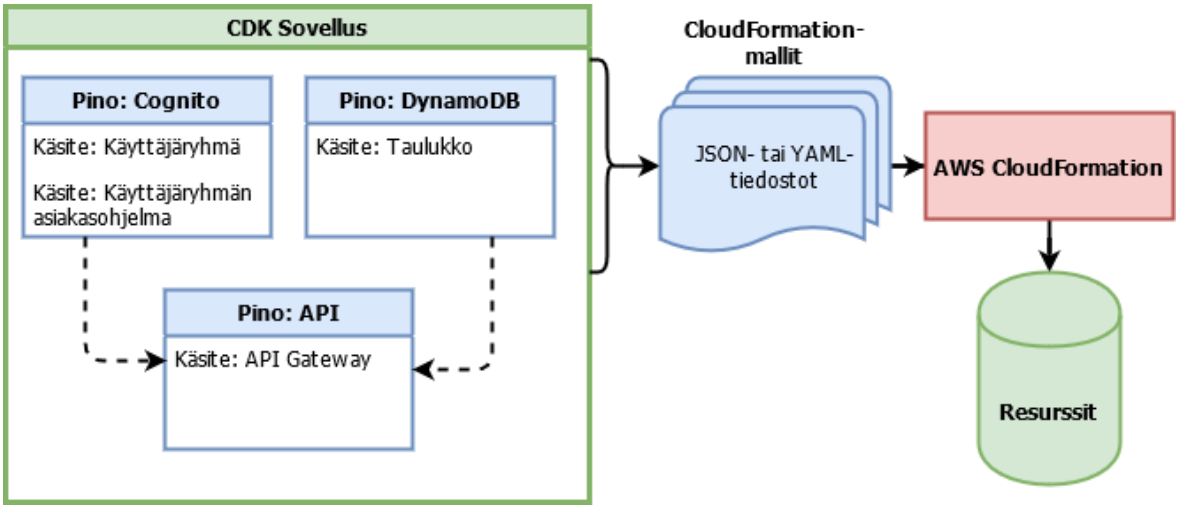
2.2.2 Cloud Formation

Cloud Formation on yksi AWS:n tarjoamista tavoista, jolla infrastruktuuri voidaan luoda koodina. Haluttu palvelu määritetään kirjoittamalla Cloud Formation -malli JSON- tai YAML-tiedostona. Kun malli on valmis, voidaan se ajaa joko CLI-työkalun läpi tai ladata se konsolissa Cloud Formation -osion kautta, jonka jälkeen uusi palvelu on käytettävissä juuri sellaisena kuin se oli määritetty. Cloud Formation -malleissa on kuitenkin yksi ongelma; ne ovat todella työläitä kirjoittaa. Palvelun tai resurssin luominen vaati suuren määrän eri ominaisuuksia määrittämistä, jotka eivät kuitenkaan ole kehityksen kannalta kovin olennaista tietoa. Pahimmillaan Cloud Formation -mallit voivat olla useita satoja rivejä pitkiä, kun taas saman resurssin luominen CDK:lla rajoittuu kymmeniin. (Serverless Stack, 2020.)

2.2.3 AWS Cloud Development Kit

Vuonna 2018 julkaistu AWS CDK (Cloud Development Kit) on Amazonin ratkaisu Cloud Formationin ongelmiin. CDK mahdollistaa infrastruktuurin määrittämisen nykyajan tutuilla ohjelmointikielillä. Alun perin tuettuna kielenä oli vain TypeScript, mutta sittemmin tukea on laajennettu JavaScript-, Python-, Java-, C#- ja Go-kieliin. Ohjelmointikielen käyttö on CDK suurin valttikortti, sillä se mahdollistaa ohjelmoinnista tuttujen konseptien kuten konditionaalien ja olio-ohjelmoinnin tekniikoiden hyödyntämisen. Koodi voidaan pilkkoa osiin ja uudelleen käyttää niitä toisissa projekteissa ja infrastruktuurin kasvaessa sen hallitseminen on helpompaa. Lisäksi sillä voidaan hyödyntää ohjelmointiympäristöjen koodin täydentämisen ominaisuuksia. (Amazon Web Services, 2021.)

CDK:lla luodut infrastruktuurit koostuvat kokonaisuudessaan sovelluksista (engl. Application), joka puolestaan sisältävät yhden tai useamman pinon (engl. Stack). Pino taas koostuu käsitteistä (engl. Construct), jotka vastaavat AWS:n eri palveluita ja resursseja, esimerkiksi Cognito-käyttäjäryhmä (engl. User Pool) tai DynamoDB. Kun haluttu infrastruktuuri on määritetty, ajetaan se CDK:n CLI-työkalun läpi, joka muodostaa koodista ensin Cloud Formation -mallin, joka ladataan AWS Cloud Formation -palveluun, joka lopuksi luo resurssit mallien mukaan (KUVA 1). Cloud Formation on siis edelleen osa prosessia, mutta suuri osa asioista, jotka jouduttiin ennen määrittämään itse, on nyt valmiiksi määritetty käsitteissä. (Amazon Web Services, 2021.)



KUVA 1. CDK:n toimintaprosessi

2.2.4 Serverless Stack

Serverless Stack ei itseasiassa ole AWS:n palvelu tai työkalu, vaan Anomaly Innovations:n luoma kirjasto, jota voidaan käyttää CDK:n rinnalla. Serverless Stack luotiin helpottamaan ja nopeuttamaan CDK-sovellusten kehitystä luomalla käyttäjälle valmiiksi pohjan projektille ja asentamalla usein tarvittavia liitännäisiä ja muita kirjastoja, jotta kehitys on nopeampi aloittaa. Serverless Stack ei kuitenkaan muuta normaalia CDK:n työskentelytapaa tai konsepteja, eikä myöskään korvaa CDK:tä, joten molempien kirjastojen moduuleja voidaan käyttää samassa projektissa niin paljon tai vähän kuin halutaan. Monet Serverless Stackin käsitteistä perivätkin vastaavan CDK:n käsitteen ominaisuudet tai sitovat pari niistä yhteen. (Serverless Stack, 2020.)

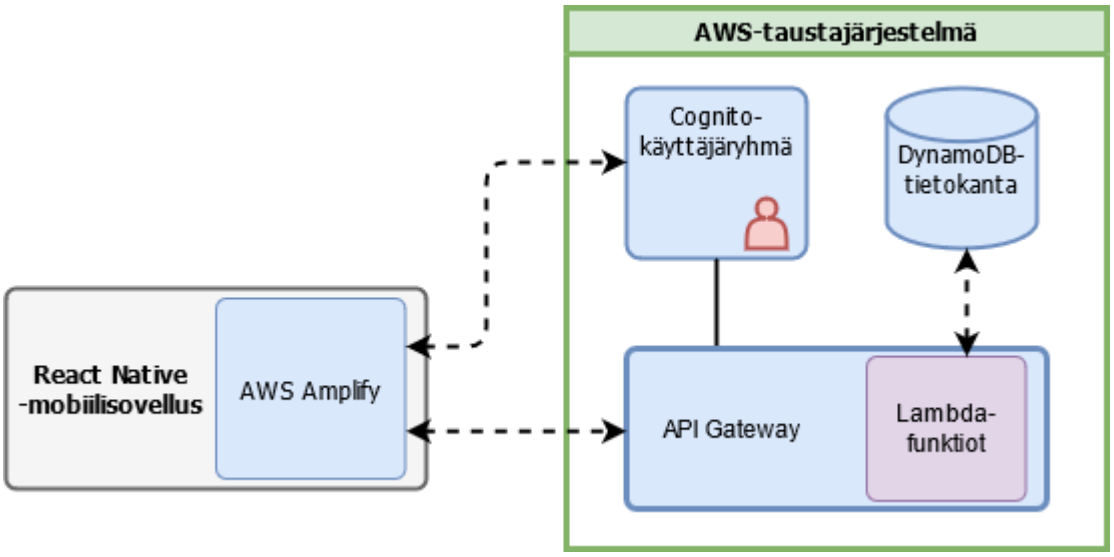
Yksi Serverless Stackin hyödyllisimmistä ominaisuuksista on niin sanottu Live Lambda Development -ympäristö. Kun CDK:llä luodaan Lambda-funktioita ja halutaan testata niiden toimintaa, uudet muutokset pitää ottaa käyttöön CDK-työkalulla, joka yleensä kestää minuutin tai kaksikin. Tämä hidastaa kehitystä pidemmällä aikavälillä huomattavasti. Toinen tapa on tehdä jäljitelmät käytetyistä palveluista, mutta tämä on kömpelöä ja sisältää muita rajoituksia. Serverless Stack tarjoaa ratkaisun tähän komennolla *sst start*, joka luo testiympäristön. Testiympäristön ollessa päällä Lambda-funktioihin tehdyt muutokset ovat heti voimassa, mikä nopeuttaa kehitystä huomattavasti. (Serverless Stack, 2020.)

3 TAUSTAJÄRJESTELMÄN TOTEUTUS

3.1 Yleiskuvaus

Taustajärjestelmä (engl. back end) toteutettiin AWS CDK- ja Serverless Stack (SST) -kirjastoilla, käyttäen JavaScript-ohjelmointikieltä. Varsinainen projekti luotiin CDK:n sijaan SST:llä, koska se antaa hyvän alkutilanteen projektille ja mahdollistaa testiympäristön pystyttämisen, josta mainittiin luvussa 2.2.4. Palveluita tullaan käyttämään mobiilisovelluksessa, joka vaatii taakseen autentikointipalvelun, tietokannan ja sitä käsittelevät toiminnot. Luodut palvelut elävät täysin AWS:n pilvipalvelussa, eikä erillisiä palvelimia luoda. Lambda-funktiot, jotka tässä tapauksessa tulevat sisältämään perinteisiä lisää-, lue-, päivitä- ja poista-toimintoja (engl. Create-, Read-, Update- ja Delete eli CRUD), ajetaan tilattomassa säiliössä. Säiliöt käynnistyvät erilaisten tapahtumien johdosta, esimerkiksi HTTP-pyynnön saapuessa tai tiedostoa ladattaessa, ja pysyvät käynnissä hetken aikaa odottamassa uusia tapahtumia, jolloin niihin voidaan vastata nopeammin. (Serverless Stack, 2020.) Lambda-funktioihin tutustutaan tarkemmin luvussa 3.4.

Tärkeimmät käytetyt palvelut ovat DynamoDB, Cognito ja API Gateway, johon voidaan lukea mukaan Lambda-funktiot. DynamoDB on NoSQL-tyylinen tietokanta, johon tullaan tallentamaan käyttäjän syöttämiä tapahtumia. Cognito hoitaa käyttäjän rekisteröinnin ja kirjautumisen, sekä auktorisoi saapuvat API-kutsut. Lopuksi API Gateway sitoo nämä kaksi palvelua yhteen ohjaamalla API-kutsut oikeaan paikkaan tai estää ne kokonaan (KUVA 2).



KUVA 2. Työn infrastruktuuri

3.2 DynamoDB

Kuten edellisessä luvussa mainittiin, DynamoDB on NoSQL-tietokanta. Tietokannan taulukot koostuvat avain-arvo-pareista ja sitä käsitellään pääasiassa oliomaisesti metodeilla sen sijaan, että suoritetaan SQL-lauseita (Amazon Web Services, 2021). Koska käytössä on ohjelmointikieli ja koska CDK-sovellukset voivat sisältää useamman pinon, luodaan DynamoDB:stä oma pino (joka on siis Ja-

vaScript-luokka), joka perii SST-kirjaston Stack-luokan. Jakamalla pinot omiin tiedostoihin niiden hallinta helpottuu, kun sovellus ajan myötä kasvaa, ja lisäksi niistä voidaan luoda uudelleenkäytettäviä moduuleja muihin projekteihin.

Pinon muodostimessa määritetään siihen kuuluvat resurssit, tässä tapauksessa tietokantaan kuuluvat taulukot. Mobiilisovellusta varten luodaan *EventTable*-niminen taulukko, jolle määritetään laskutustapa ja pääavain. Annettu laskutustapa kertoo DynamoDB:lle kuinka haluamme sen käytöstä maksaa, tässä tapauksessa jokaisesta luku- ja kirjoitusoperaatiosta. Lisäksi taulukolle määritetään pääavaimen nimeksi *eventId* ja sen tyypiksi merkkijono. Taulukolle luodaan myös toissijainen indeksi, jotta sen tiedot saadaan helpommin haettua aikajärjestyksessä. Lopuksi esitellään jäsenmuuttuja *tables*, johon sijoitetaan tieto taulukosta, koska sitä tullaan tarvitsemaan API Gateway:ta luotaessa. Alla olevasta kuvasta nähdään (KUVA 3), että vaikka projekti ja pino luotiin SST:llä, voidaan muodostimessa silti käyttää vapaasti CDK:n kirjastoja, josta Table-luokka on peräisin.

```
export default class DynamoDBStack extends sst.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const app = this.node.root;

    const eventTable = new dynamodb.Table(this, 'EventTable', {
      billingMode: dynamodb.BillingMode.PAY_PER_REQUEST,
      partitionKey: { name: 'eventId', type: dynamodb.AttributeType.STRING },
    });

    eventTable.addGlobalSecondaryIndex({
      indexName: 'userId',
      partitionKey: { name: 'userId', type: dynamodb.AttributeType.STRING },
      sortKey: { name: 'date', type: dynamodb.AttributeType.NUMBER },
    });

    this.tables = {
      event: eventTable,
    };
  }
```

KUVA 3. Ote DynamoDB:n määrittävästä koodista

3.3 Cognito

Cognito on autentikointipalvelu käyttäjien rekisteröinnin ja kirjautumisen hallintaan ja toteutukseen sovelluksissa, lisäksi se voi toimia auktorisoijana muihin AWS:n resursseihin. Cognito tukee myös tunnistautumista sosiaalisen median palvelujen tileillä, kuten Google tai Facebook. (Amazon Web Services, 2021.) Tässä työssä vaihtoehtoisia kirjautumisvaihtoehtoja ei kuitenkaan käytetty.

Cognito-pinon toteutus ei poikkea muista edellisestä käytettyjä resursseja lukuun ottamatta, eli ensin esitetään uusi luokka, joka perii SST-kirjaston Stack-luokan ominaisuudet ja sen jälkeen määritetään halutut resurssit muodostimessa (KUVA 4). Cognitoa varten on luotava uusi käyttäjäryhmä, jonne rekisteröityneet käyttäjät tallennetaan. Käyttäjäryhmä mahdollistaa myös kirjautumisen ja rekisteröitymisen. Oletuksena ylläpitäjän täytyisi manuaalisesti hyväksyä uudet käyttäjät, minkä takia *selfSignUpEnabled* asetetaan päälle. Uudet käyttäjät tulevat saamaan varmistuskoodin määrityksistä

riippuen joko sähköpostiin tai puhelimeen tekstiviestinä, jonka he syöttävät sovellukseen rekisteröitymisvaiheessa. Varmistuskoodin ja myös salasanan palauttamisen takia käyttäjiin on hyvä olla jokin tapa ottaa yhteyttä, ja siksi käyttäjätunnuksena käytetään sähköpostiosoitetta. Rekisteröinti voitaisiin hoitaa myös tavallisella käyttäjätunnuksella ja sen lisäksi pyytää käyttäjää syöttämään sähköposti tai puhelinnumero yhteystiedoksi.

Autentikointia varten on luotava vielä käyttäjäryhmän asiakasohjelma (engl. User Pool App Client), jolle on määritettävä ainakin yksi identiteetin tarjoaja. Tässä tapauksessa riittää, että tarjoajaksi määritetään äsken luotu käyttäjäryhmä. Mikäli sovelluksessa olisi mahdollista käyttää vaihtoehtoisia kirjautumistapoja, määritettäisiin ne myös tässä. Lopuksi luodut resurssit sijoitetaan jäsenmuuttujaan *resources*, jotta niihin päästään käsiksi API Gateway:ta, määritettäessä.

```
const userPool = new cognito.UserPool(this, 'UserPool', {
  selfSignUpEnabled: true,
  signInAliases: { email: true },
  signInCaseSensitive: false,
});

const userPoolClient = new cognito.UserPoolClient(this, 'UserPoolClient', {
  userPool,
  generateSecret: false,
});

this.resources = {
  userPool,
  userPoolClient,
}
```

KUVA 4. Ote Cognito:n määrittävästä koodista

3.4 API Gateway

API Gateway -palvelu ohjaa ja hallinnoi tulevia pyyntöjä REST-, HTTP- ja WebSocket-päätepisteisiin ja toimii siis eräänlaisena porttina palveluihin (Amazon Web Services, 2021). Pinon määritys eroaa aiemmista siten, että luokka saa parametreinä DynamoDB-taulukot ja Cognito-resurssit (KUVA 5).

```
export default function main(app) {
  const dynamoDbStack = new DynamoDBStack(app, 'dynamodb');
  const cognitoStack = new CognitoStack(app, 'cognito');
  new ApiStack(app, 'api', dynamoDbStack.tables, cognitoStack.resources);
}
```

KUVA 5. API Gateway:lle annetaan tiedot muista palveluista

API Gateway:n luomiseen käytetään SST-kirjaston Api-luokkaa. Auktorisoijaksi määritetään aiemmin luotuun käyttäjäryhmään liittyvät resurssit. Saapuvat pyynnöt täytyy tulla rekisteröityneeltä käyttäjältä, eli käyttäjätilin on oltava olemassa käyttäjäryhmässä. Auktorisoinnin tyypiksi määritetään JWT, jolloin saapuvien HTTP-pyyntöjen Header-osiossa on oltava validi JWT-merkki. JWT-tyyppiä voi käyt-

tää vain API-päätepisteiden turvaamiseen, mutta koska muita turvattavia AWS-resursseja ei ole, sopii se tässä tapauksessa hyvin. API Gateway:lle annettava myös lupa käyttää aiemmin luotua Event-taulukkoa (KUVA 6).

API-kutsuja varten määritetään lopuksi reitit eri Lambda-funktioihin. ”Lambda” on hieno nimitys funktiolle, joka suoritetaan jonkin tapahtuman johdosta. Tässä tapauksessa tapahtumat ovat HTTP-pyyntöjä. Reitit muodostuvat arvo-avain-pareista, jossa arvo sisältää HTTP-metodin ja välilyönnillä erotettuna loppuosan URL:sta. Arvona on polku projektihakemiston juuresta tiedostoon, jossa Lambda-funktio on määritetty, sekä pisteen jälkeen funktion nimi. Arvo on siis muotoa *{hakemisto}/{tiedosto}.{funktio}* (KUVA 6).

```
export default class ApiStack extends sst.Stack {
  constructor(scope, id, dbTables, cognitoRes, props) {
    super(scope, id, props);

    const { userPool, userPoolClient } = cognitoRes;

    const api = new sst.Api(this, 'Api', {
      defaultAuthorizer: new apigwAuthorizers.HttpUserPoolAuthorizer({
        userPool,
        userPoolClient,
      }),
      defaultAuthorizationType: sst.ApiAuthorizationType.JWT,
      defaultFunctionProps: {
        environment: {
          tableName: dbTables.event.tableName,
        }
      },
      routes: {
        'POST /event/add': 'src/add.handler',
        'POST /event/modify/{id}': 'src/modify.handler',
        'POST /event/delete/{id}': 'src/delete.handler',
        'GET /event/get': 'src/getAll.handler',
        'GET /event/get/{id}': 'src/getById.handler',
      },
    });

    api.attachPermissions([dbTables.event])
```

KUVA 6. Ote API Gateway:n määrittävästä koodista

Lambda-funktio voi näyttää esimerkiksi alla olevan kuvan mukaiselta (KUVA 7). Funktio saa parametrina *event*-objektin, josta poimitaan käyttäjän syöttämät tiedot sekä käyttäjän Id-tunnisteen. Tiedoista muodostetaan taulukkoon sopiva objekti, joka annetaan parametrina taulukon nimen lisäksi *put()*-metodille, joka lisää uuden alkion tietokantaan. Lopuksi käyttäjälle palautetaan vastaus suoritetun operaation tuloksesta.

```
export async function handler(event) {
  const { date, sport, duration, distance, comment } = JSON.parse(event.body);
  const userId = event.requestContext.authorizer.jwt.claims.sub;

  const itemForDb = {
    eventId: uuidv4(),
    userId,
    date,
    sport,
    duration,
    distance,
    comment,
  };

  await dynamoDb
    .put({
      TableName: process.env.tableName,
      Item: itemForDb,
    })
    .promise();

  return {
    statusCode: 200,
    body: JSON.stringify(itemForDb),
    headers: { 'Content-Type': 'application/json' },
  };
```

KUVA 7. Esimerkki Lambda-funktiosta

3.5 Käyttöönotto

Kun infrastruktuuri on määritetty, täytyy se ottaa käyttöön. Kuten on mainittu, infrastruktuuri koodina -mallin vahvuuksiin kuuluu käyttöönoton helppous. SST sekä CDK tekee tästä todella helppoa, myös silloin, kun infrastruktuurista halutaan luoda eri ympäristöjä, kuten kehitys- ja tuotantoympäristöt. Kun projekti ensimmäisen kerran luotiin, SST teki projektihakemiston juureen *sst.json*-nimisen tiedoston (KUVA 8).

```
{
  "name": "urheiluseuranta-aws",
  "stage": "dev",
  "region": "eu-central-1",
  "lint": true
}
```

KUVA 8. Projektin oletusasetukset

Tässä tiedostossa kerrotaan oletusasetukset infrastruktuurin käyttöönotolle. Tärkeimmät kentät tiedostossa ovat *stage* ja *region*. Stage-parametrillä kerrotaan ympäristön nimi, tässä tapauksessa *dev* (development eli kehitys), ja region-parametri määrittää missä päin maailmaa infrastruktuuri tulee sijaitsemaan. Ympäristö on vapaasti nimettävissä, mutta alueen täytyy vastata jotain AWS:n tukemaa sijaintia. (Serverless Stack, 2020.)

Infrastruktuuri otetaan käyttöön määritetyillä asetuksilla seuraavalla komennolla (KUVA 9):

```
❯ yarn sst deploy
```

KUVA 9. Käyttöönotto oletusasetuksilla

Jos infrastruktuuri halutaan ottaa käyttöön eri ympäristössä tai maa-alueella, voidaan oletusasetusten arvot korvata alla olevan komennon mukaisesti (KUVA 10):

```
❯ yarn sst deploy --stage prod --region eu-west-1
```

KUVA 10. Oletusasetusten ylikirjoitus

Käyttöönoton jälkeen resurssit ovat tarkasteltavissa ja myös muokattavissa AWS:n konsolissa samalla tavoin kuin jos ne olisi luotu manuaalisesti. Alla olevassa kuvassa (KUVA 11) nähdään miltä SST:n luomat pinot voivat näyttää Cloud Formation -osiossa. Pinojen nimet koostuvat ympäristön ja sovelluksen nimistä, jotka määritettiin *sst.json*-tiedostossa tai käyttöönottokomennon argumenteissa, sekä resurssin nimestä.

Stack name
prod-urheiluseuranta-aws-api
prod-urheiluseuranta-aws-cognito
prod-urheiluseuranta-aws-dynamodb
dev-urheiluseuranta-aws-api
dev-urheiluseuranta-aws-dynamodb
dev-urheiluseuranta-aws-cognito

KUVA 11. Esimerkki luoduista pinoista

4 LUOTUJEN PALVELUIDEN KÄYTTÖ

4.1 AWS Amplify

Tässä luvussa tullaan tarkastelemaan kuinka pystytetyn infrastruktuurin resursseja voidaan käyttää sovelluksessa. Resurssien toiminta todetaan hyödyntämällä niitä React Native -ohjelmistokehyksellä toteutetussa mobiilisovelluksessa, ja tätä varten AWS Amplify on varteenotettava vaihtoehto, sillä se nivoutuu hyvin yhteen muiden AWS:n palveluiden kanssa. AWS Amplify on kokoelma työkaluja, kirjastoja ja käyttöliittymäkomponentteja, joita voidaan käyttää yhdessä tai erikseen web- ja mobiilisovellusten kehitykseen ja helposti yhdistämään sovellus ja taustajärjestelmä toistensa kanssa. (Amazon Web Services, 2021.)

Amplify on ensin konfiguroitava, jotta haluttuja resursseja voidaan käyttää. Tämä tehdään yleensä sovelluksen käynnistyksen yhteydessä, koska sovellus voi tarvita tietoja esimerkiksi käyttäjän edellisen kirjautumisistunnon palauttamiseen. React Native -sovelluksissa konfigurointi tehdään yleensä *index.js*- tai *App.js*-tiedostossa *configure()*-metodilla, joka ottaa parametrikseen resurssien tiedot JSON-muodossa. Ylläpidon helpottamiseksi tiedot voidaan kirjoittaa erilliseen JavaScript-kirjastoon objektina, mutta erilaisten tuotanto- ja kehitysympäristöjen määrän kasvaessa oikean tiedoston käyttö voi olla hieman kömpelöä ja se vaatii muutoksia sovelluksen koodiin. Sen sijaan React Native -sovelluksissa parempi vaihtoehto on käyttää react-native-config-kirjastoa, jonka avulla tarvittavat tiedot voidaan lisätä sovelluksen ympäristömuuttujiin. Ympäristöjä varten luodaan omat tiedostot, kuten *.env.development* ja *.env.production*, joihin viitataan sovellusta ajaessa riippuen siitä, mihin ympäristöön sovelluksen halutaan yhdistävän.

4.2 Autentikointi

Sovelluksen käyttöä varten käyttäjän on pystyttävä luomaan uusi käyttäjätunnus ja kirjautumaan sovellukseen. Tätä varten Amplify:n konfiguraatioon on lisättävä Auth-osio, jossa kerrotaan maa-alue, jonne Cognito-palvelu on pystytetty sekä käyttäjäryhmän tunnisteet (KUVA 12). Tunnistetiedot löytyvät AWS konsolin Cognito-osiosta sen jälkeen, kun palvelu on otettu käyttöön. Perustietojen lisäksi kirjautuminen määritetään pakolliseksi.

```
Amplify.configure({
  Auth: {
    mandatorySignIn: true,
    region: awsConfig.cognito.REGION,
    userPoolId: awsConfig.cognito.USER_POOL_ID,
    userPoolWebClientId: awsConfig.cognito.APP_CLIENT_ID,
  },
```

KUVA 12. Autentikoinnin konfiguraatio

Varsinaisten autentikointitoimintojen toteutus Amplify:llä on hyvin yksinkertaista. Tarvittavat metodit löytyvät Auth-luokan alta ja ne hoitavat kaiken monimutkaisen logiikan kehittäjän puolesta. Kehittäjän tehtäväksi jää parametrien välittäminen metodille ja paluuarvon käsittely. Esimerkiksi kirjautuminen toteutetaan välittämällä käyttäjän syöttämä käyttäjätunnus ja salasana *signIn()*-metodille. Jos kirjautuminen onnistui, käyttäjä ohjataan päänäkymään tai jos se epäonnistui, käyttäjälle näytetään virheilmoitus (KUVA 13).

```
AuthService.signIn(username, password)
  .then(() => {
    setIsAuthed(true); // Käyttäjä ohjataan päänäkymään
  })
  .catch((error) => {
    Alert.alert(error.message); // Käyttäjälle näytetään virheilmoitus
  });
```

KUVA 13. Sisäänkirjautumisen käsittelevä funktio

4.3 CRUD-operaatiot

Lisää-, lue-, päivitä- ja poista-toimintojen toteutuksessa on pari vaihtoehtoa, joilla HTTP-pyynnöt taustajärjestelmälle voidaan lähettää. Toimintojen suoritukseen ei tarvita Amplify:a, vaan HTTP-pyynnöt voidaan lähettää esimerkiksi suositulla Axios-kirjastolla tai JavaScriptin omalla *fetch()*-metodilla, mutta Amplify sisältää ominaisuuksia, jotka auttavat toimintojen toteutuksessa. Amplify-kirjasto sisältää API-nimisen luokan, jolla HTTP-pyyntöjä voidaan lähettää kuten Axios-kirjastolla. API-luokka itse asiassa käyttää Axios-kirjastoa pyyntöjen lähettämiseen ja koska Amplify-kirjastoa tarvittiin autentikoinnin toteuttamiseen, päädyttiin opinnäytetyössä käyttämään API-luokkaa.

HTTP-pyyntöjen suorittamiseen tarvittavat tiedot voidaan konfiguroida *configure()*-metodilla, kuten edellisessä luvussa tehtiin autentikointia toteuttaessa (KUVA 12). API-luokan konfigurointiin täytyy määrittää vähintäänkin nimi sekä API Gateway:n URL-osoite ja maa-alue. Näiden lisäksi konfiguraatioon määritetään *custom_header*-attribuutti. Luvussa 3.4 API Gateway:n auktorisoinnin tyypiksi määritettiin JWT, joten HTTP-pyyntöjen mukana on lähetettävä validi JWT-merkki, jotta Lambda-funktioita voitaisiin suorittaa. Kirjautuneen käyttäjän JWT-merkki voidaan hakea käyttämällä apuna Auth-luokan metodeja ja siten sisällyttää se konfiguraation *custom_header*-attribuuttiin (KUVA 14).

```
API: {
  endpoints: [
    {
      name: 'Api',
      endpoint: awsConfig.apiGateway.URL,
      region: awsConfig.apiGateway.REGION,
      custom_header: async () => {
        return {
          Authorization: `Bearer ${(await Auth.currentSession())
            .getIdToken()
            .getJwtToken()}`,
        };
      },
    },
  ],
}
```

KUVA 14. API-luokan konfiguraatio

API-luokan metodien käyttö on hyvin yksinkertaista. Alla olevassa kuvassa (KUVA 15), lähetetään käyttäjän muokkaaman tapahtuman uudet tiedot taustajärjestelmään tietokantaan tallennettavaksi. API-luokan *post()*-metodille riittää, että sille välitetään API-päätepisteen nimi, URL-osoitteen loppuosa sekä tapahtuman uudet tiedot. Luokan metodit noutavat muut tiedot Amplify:n konfiguraatiosta, mikä vähentää saman koodin toistamista. Parametreissä API_NAME on muuttuja, johon on sijoitettu päätepisteen nimi, eli tässä tapauksessa ”Api”, niin kuin yläpuolella määritettiin (KUVA 14). Jotta tiedetään mitä tietokannan alkiota on muokattava, sisällytetään URL-osoitteeseen tapahtuman Id-tunniste.

```
await API.post(API_NAME, `/event/modify/${event.eventId}`, params)
```

KUVA 15. Esimerkki HTTP-pyynnön lähettämisestä

5 POHDINTA

Opinnäytetyötä tehdessä tuli tarkoituksenmukaisesti hyvin tutuksi CDK:n keskeisimmät konseptit ja sen kanssa käytetyt tekniikat ja toimintatavat. Infrastruktuuri koodina -osiota kirjoittaessa asiasta tuli esille useita asioita, joita en ennen olisi osannut huomioida, vaikka ne jälkikäteen tuntuvatkin hyvin itsestään selviltä. Erityisesti infrastruktuurin tilan toistettavuuteen ja dokumentaation yhtenäisyyteen liittyvät seikat olivat hyvin silmiä avaavat. Kaikkiaan työ antoi hyvän pohjan tulevaisuutta varten.

Serverless Stack (SST) oli täysin uusi asia, jonka otin opinnäytetyöhön mukaan katsoakseni olisiko se hyödyllinen kirjasto infrastruktuurin kehityksessä ja myös mielenkiinnon vuoksi. SST mahdollisti kehityksen aloittamisen käytännössä heti ilman liikoja valmisteluja eikä se muuta CDK:n normaaleja työskentelytapoja, joten CDK:n dokumentaatiota pystyi helposti soveltamaan SST:n kanssa. Hyödyllisin ominaisuus oli ehdottomasti testiympäristö, joka nopeuttaa työtä huomattavasti, ei vaadi ylimääräistä valmistelua toimiakseen ja tekee työstä myös mielekkäänpää. SST:n isoimmat hyödyt piilevätkin projektin aloituksen nopeuttamisessa, ja ei siksi ehkä ole kovin houkutteleva kokeneelle CDK-osaajalle edes testiympäristön vuoksi, joka on kuitenkin toteutettavissa muillakin tavoin. Lisäksi se on yksi kirjasto lisää ylläpidettäväksi kaiken muun joukossa.

Suurin haaste opinnäytetyössä oli infrastruktuurin kehityksen aloittaminen ja joissain määrin AWS:n ja CDK:n dokumentaation laadun vaihtelevuus. Vaikka SST luokin hyvän pohjan projektille, ei kehitystä varsinaisesti pystynyt aloittamaan ennen kuin olin perehtynyt kunnolla CDK:n konsepteihin, dokumentaatioon ja työssä käytettyjen palveluiden ominaisuuksiin ja asetuksiin. Jopa kaiken lukemisen jälkeen koodia täytyi toisinaan kirjoittaa hieman sokkona, etenkin kun palvelut halusi yhdistää toisiinsa, koska palvelujen toimivuutta ei pystynyt heti varmistamaan. Ohjeita soveltamalla sekä CLI-työkalujen ja Postman-ohjelman avulla sain kuitenkin palvelut toimimaan halutulla tavalla yllättävänkin nopeasti.

LÄHTEET

Red Hat julkaisuaika tuntematon. *What is Infrastructure as Code (IaC)?* [Viitattu: 20.4.2021] https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac

IBM Cloud Education 2019. *Infrastructure as Code*. [Viitattu: 20.4.2021] https://www.ibm.com/cloud/learn/infrastructure-as-code

Schults, Carlos 2019. *Stackify. What Is Infrastructure as Code? How It Works, Best Practices, Tutorials*. [Viitattu: 20.4.2021] https://stackify.com/what-is-infrastructure-as-code-how-it-works-best-practices-tutorials/

Amazon Web Services 2021. *AWS Cloud Development Kit Documentation*. [Viitattu 9.4.2021] https://docs.aws.amazon.com/cdk/index.html

Amazon Web Services 2021. *Amazon DynamoDB Documentation*. [Viitattu: 9.4.2021] https://docs.aws.amazon.com/dynamodb

Amazon Web Services 2021. *Amazon Cognito Documentation*. [Viitattu: 14.3.2021] https://docs.aws.amazon.com/cognito

Amazon Web Services 2021. *Amazon API Gateway Documentaion*. [Viitattu: 9.4.2021] https://docs.aws.amazon.com/apigateway

Amazon Web Services 2021. *Amplify Docs. Authentication.* [Viitattu: 11.4.2021] https://docs.amplify.aws/lib/auth/getting-started/q/platform/js

Amazon Web Services 2021. *Amplify Docs. API (REST)*. [Viitattu: 11.4.2021] https://docs.amplify.aws/lib/restapi/getting-started/q/platform/js

Amazon Web Services 2021. *Security best practices in IAM*. [Viitattu: 9.4.2021] https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html

Serverless Stack 2020. *Serverless Stack Guide*. [Viitattu: 20.4.2021] https://serverless-stack.com/

Serverless Stack 2020. *Deploying Your App*. [Viitattu: 2.5.2021] https://docs.serverless-stack.com/deploying-your-app

Serverless Stack 2020. *Live Lambda Development*. [Viitattu 20.4.2021] https://docs.serverless-stack.com/live-lambda-development