Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

AARO PENNANEN

REST API:n siirtäminen serverless-ympäristöön

TIETOJENKÄSITTELYN TUTKINTO-OHJELMA
2023

TIIVISTELMÄ

Pennanen, Aaro: REST API:n siirtäminen serverless-ympäristöön
Opinnäytetyö, AMK
Tietojenkäsittely
Marraskuu 2023
Sivumäärä: 38

Tämän opinnäytetyön tarkoituksena oli tutkia serverless-arkkitehtuuria ja sen soveltamista .NET-ympäristössä. Ensimmäiseksi työssä tutustutaan serverless-arkkitehtuuriin ja sen ominaisuuksiin. Toisessa osassa käydään lyhyesti läpi C# ohjelmointikieli ja .NET Core-ohjelmistokehys, sekä esitellään REST-API, joka oli toteutettu .NET Core:lla ja C# ohjelmointikielellä.

Tämän jälkeen syvennyttiin konkreettisiin koodimuutoksiin, jotka mahdollistavat API:n toiminnan eri palveluntarjoajien serverless-ympäristöissä. Lisäksi työssä tarkasteltiin siirtoprosessia, missä perinteisessä ympäristössä oleva REST-API siirrettiin serverless-ympäristöihin, sekä käytiin läpi tarvittavien serverless-konfiguraatioiden määrittelyä, joka on oleellinen osa serverless-arkkitehtuuria.

Lopuksi opinnäytetyössä pohditaan siirtoprosessin tuloksia ja onnistumista. Tuloksista ilmeni, että REST-API:n siirtäminen serverless-ympäristöihin, joka oli tämän opinnäytetyön päätarkoituksena, onnistui hyvin. Serverless-ympäristöön siirretty API toimi odotetulla tavalla, vaikka koodiin ja infrastruktuuriin oli tehty muutoksia. Tuloksissa pohdittiin myös serverless-ympäristöihin siirtymiseen liittyviä haasteita esimerkiksi mahdollisen autentikoinnin tai tiedostonkäsittelyn toteuttamisen osalta.

Avainsanat: Serverless, C#, .NET CORE, Web-kehitys, AWS Lambda, Google Cloud Functions, Azure Functions, REST-API

Abstract

Pennanen, Aaro: Migrating a REST-API to serverless environment
Bachelor's thesis
Bachelor of Business Information Systems
November 2023
Number of pages: 38

The purpose of this thesis was to investigate serverless architecture and its application in a .NET environment. First, the work introduces the features of serverless architecture. In the second part, the C# programming language and the .NET Core software framework are briefly introduced, followed by an introduction to the REST API, which was implemented with .NET Core and the C# programming language.

After this, the thesis delves into the specific code changes that enables the API functionality across various serverless environments provided by different service providers. In addition, the work examines the migration process in which a REST API in a traditional environment is transferred to serverless environments, as well as the management of necessary serverless configurations, which are an essential part of serverless architecture.

Finally, the results and success of the transfer process are discussed. The results showed that the transfer of the REST API to serverless environments, which was the main purpose of this thesis, was successful. The API transferred to the serverless environment worked as expected, even though changes had been made to the code and infrastructure. The results also addressed challenges associated with migrating to serverless environments, such as the implementation of possible authentication or file handling.

Keywords: Serverless, C#, .NET CORE, Web-development, AWS Lambda, Google Cloud Functions, Azure Functions, REST-API

SISÄLLYS

SYMBOLI- JA LYHENNELUETTELO

REST Representional State Transfer on arkkitehtuurimalli, jolla määritellään, miten sovelluksen dataa käsitellään, kysellään ja muokataan.

API Application Programming Interface on rajapinta, jolla voidaan kommunikoida ja siirtää tietoa eri sovellusten välillä.

ORM Object Relational Mapping on ohjelmointitekniikka, jonka avulla tietokannan taulut voidaan muuntaa olioksi ja toisinpäin.

CLI Command Line Interface on työkalu, jolla voidaan syöttää tietokoneelle tai järjestelmälle tekstimuotoisia komentoja.

Riippuvuusinjektio (Dependecy injection) on ohjelmoinnin tekniikka, jossa komponenttien tarvitsemat riippuvuudet annetaan niille niiden ulkopuolelta.

DTO (Data Transfer Object) on olio, jolla siirretään dataa eri ohjelmakomponenttien ja kerrosten välillä.

1 JOHDANTO

IT-alan jatkuvasti kasvaessa, tulee sen myötä myös uusia vaihtoja ja ratkaisuja IT-palveluiden kehittämiselle ja ylläpidolle jatkuvasti tarjolle. Digitaalisuus alkaa olemaan jo melkein standardi sovelluksissa ja palveluissa. Fyysistä palvelimista on siirrytty virtuaalikoneisiin, ja virtuaalikoneista edelleen pilvipalveluihin. Viime vuosina yhä useammat yritykset ovat siirtyneet käyttämään pilvipalveluita.

Pilvipalveluiden avulla ylläpito- ja hallintakuluja voidaan pienentää. Pilvipalvelussa on useita eri palveluratkaisuja ja -malleja, jotka skaalautuvat joustavasti asiakkaiden ja yritysten eri tarpeisiin. Serverless-arkkitehtuuri ja siihen kuuluva Function-as-a-Service (FaaS)-palvelumalli on yksi pilvipalveluiden palvelumalleista. Kiinnostus serverless-arkkitehtuuriin on jatkuvassa kasvussa.

Tässä opinnäytetyössä käydään läpi aluksi serverless-arkkitehtuurin FaaS-palvelumallia, sekä serverless-arkkitehtuurin tärkeimpiä käsitteitä, jonka jälkeen punnitaan yleisesti serverless-arkkitehtuurin hyötyjä ja haittoja. Tämän jälkeen perehdytään tässä opinnäytetyössä käytettävään C# ohjelmointikieleen ja siihen kuuluvaan .NET alustaan.

Seuraavaksi esitellään C# ohjelmointikielellä toteutettu REST-API ja käydään läpi sen toimintoja ja ominaisuuksia. Sitten siirrytään opinnäytetyön pääaiheeseen, jossa tutkitaan miten ja millaisin muutoksin olemassa olevan C#-ohjelmointikielellä toteutetun REST-API:n saadaan vietyä kolmen suurimman pilvipalveluntarjoajan (Azure Functions, AWS Lambda, Google Cloud Functions) serverless-ympäristöihin. Lopuksi opinnäytetyössä käydään läpi implementaation onnistumista ja tuloksia.

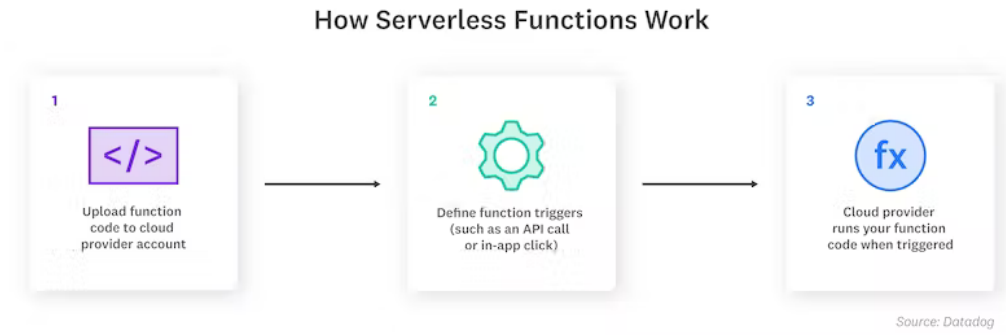
1.1 Tekoälyn käyttö tässä opinnäytetyössä

Tässä opinnäytetyössä olen käyttänyt ChatGPT:tä kielentarkistuksen apuvälineenä Englanninkielisen tiivistelmän kirjoittamiseen.

2 SERVERLESS-ARKKITEHTUURI

Vaikka termi serverless viittaa siihen, että kyseisessä arkkitehtuurissa ei käytettäisi ollenkaan palvelimia, käytännössä tämä kuitenkin tarkoittaa sitä, että ohjelmakoodi suoritetaan palveluntarjoajan palvelimella (Lintilä, 2017). Infrastruktuuri vuokrataan palveluntarjoajalta, jolloin palveluntarjoaja vastaa esimerkiksi palvelinten konfiguraatiosta, ylläpidosta, tietoturvallisuudesta ja käyttöjärjestelmästä (Onrego, n.d.). Tämä helpottaa ohjelmistokehityksen prosessia, koska vastuu infrastruktuurista on siirretty palveluntarjoajalle, jolloin ohjelmistokehityksessä voidaan keskittyä enemmän varsinaiseen koodiin ja ohjelman logiikkaan.

Serverless-arkkitehtuurin toiminta on tapahtumapohjaista, serverless-alustalla olevan sovelluksen suorittamia tapahtumia kutsutaan funktioiksi. Tätä palvelumallia kuvataan yleensä termillä Function-as-a-service tai FaaS. Funktio on ohjelmassa suoritettava koodin osa, ja funktiolle on määritelty heräte (Trigger) (Datadog, n.d.). Herätteet ovat erikseen määriteltyjä tapahtumia, jotka suorittavat funktion. Tapahtuma voi olla esimerkiksi ajastettu tapahtuma, HTTP-kutsu tai vaikkapa saapuva sähköposti.



Kuva 1. Serverless-funktion toiminta (Datadog)

Kuva 1. havainnollistaa serverless-funktioiden toiminnan, koodin logiikka on jaettu funktioihin ja funktiolle on määritelty heräte, joka suorittaa funktion palveluntarjoajan ympäristössä.

Serverless-sovellus on koko ajan palveluntarjoajan ympäristössä, ja sovellus käyttää resursseja vain silloin, kun se suorittaa funktiota, toisin kun muissa pilvipalvelun palvelumalleissa, jossa resurssit voivat olla jatkuvassa käytössä myös silloin kun sovellusta ei käytetä.

2.1 Serverless-arkkitehtuurin hyödyt

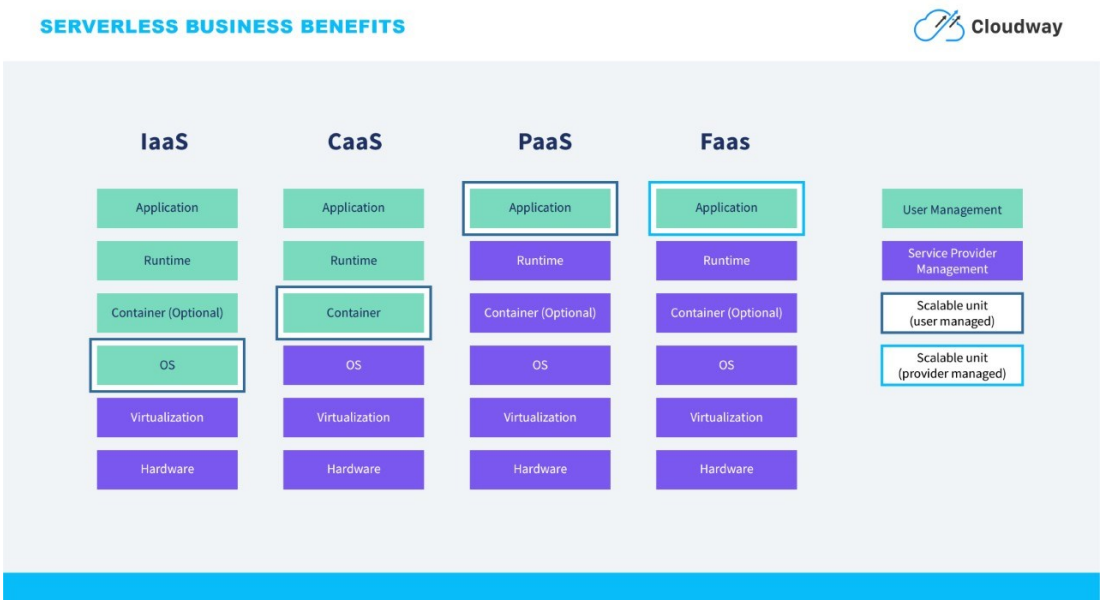
Serverless-arkkitehtuuri tarjoaa mahdollisuuden kehittää sovelluksia, ilman tarvetta ylläpitää ja määritellä infrastruktuuria. Nämä ovat serverless-arkkitehtuurin hyviä ominaisuuksia, mikä tekee serverless-arkkitehtuurista suositun vaihtoehdon sovelluskehitykselle.

2.1.1 Kustannussäästöt

Yleensä pilvipalvelut veloittavat kiinteähintaisesti käytössä olevista resursseista, esimerkiksi tallennustilasta, virtuaalikoneista ja palvelimista kuukausittain. Serverless-palveluissa palveluntarjoajat veloittavat vain resurssien käytön mukaan (Astriani, 2020).

Tämä tarkoittaa sitä, että kaikki kustannukset serverless-palveluiden käytöstä syntyvät vain käytetyistä resursseista, eli suoritettujen funktioiden määrästä, sekä niiden suoritukseen kuluneesta ajasta. Ajanjaksoilta, jolloin funktiota ei suoriteta ei myöskään veloiteta palvelun käyttäjää. Andrea Passwaterin mukaan tämän ansiosta palvelinkustannuksissa voidaan säästää jopa 70–90 % (Passwater, 2018).

Koska palveluntarjoaja vastaa palvelimista ja infrastruktuurista, serverless-sovellukset ovat helppoja ylläpitää. FaaS-palvelumallissa käyttäjä vastaa itse vain sovelluksesta (kuva 2.), ja kaikki muu infrastruktuuri on palveluntarjoajan vastuulla. Tämä myös nopeuttaa sovellusten kehitys- ja ylläpitotyötä, koska kehittäjät vastaavat itse vain sovelluksen kehittämisestä ja päivittämisestä, ilman tarvetta huolehtia infrastruktuurin ylläpidosta.



Kuva 2. Pilvipalvelumallien infrastruktuurin hallinta (Cloudway,2021)

2.1.2 Skaalautuvuus

Yksi serverless-arkkitehtuuriin liittyvistä eduista on automaattinen skaalautuvuus. Skaalautuvuus on olennainen tekijä sovelluksen tehokkaassa toiminnassa, erityisesti tilanteissa, joissa sovelluksen käyttäjämäärä kasvaa tai käyttömäärä lisääntyy odottamattomasti. Automaattinen skaalautuvuus tarkoittaa, että järjestelmä kykenee dynaamisesti sopeutumaan muuttuviin kuormituksiin.

Sovellukseen lisätään automaattisesti tarvittaessa lisää resursseja, esimerkiksi lisää funktioinstansseja, tai lisää palvelintilaa. Tällä eliminoidaan riski siitä, että palvelu ylikuormittuu (Cloudflare, n.d.b). Perinteisellä palvelimella oleva sovellus voi ylikuormittua, jos sen resurssit eivät pysty käsittelemään esimeriksi äkillisesti tai odottamattomasti kasvaneita HTTP-pyyntöjen määriä.

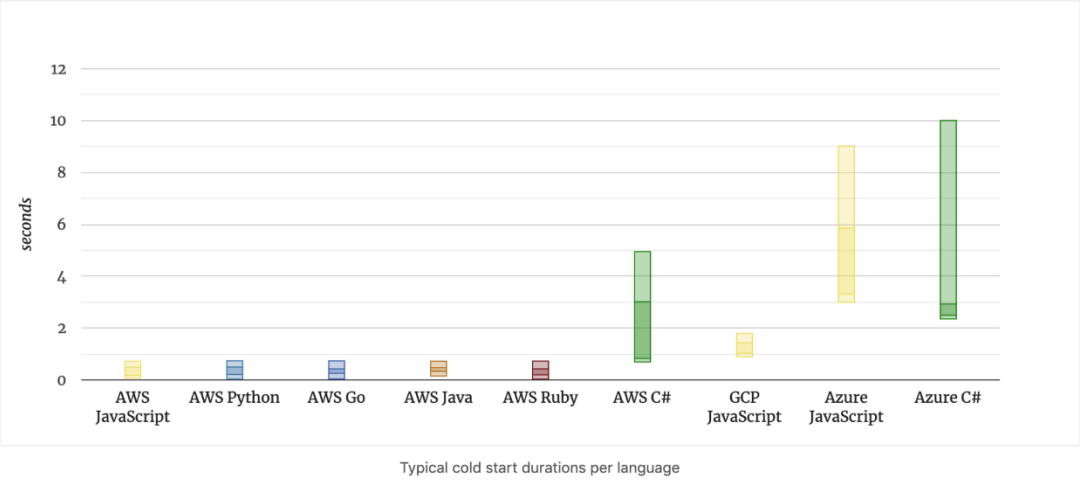
Automaattinen skaalautuvuus toimii myös toiseen suuntaan- jos sovelluksen käyttö vähenee, sovellus skaalautuu automaattisesti myös alaspäin ja sovellukselle varattuja resursseja vähennetään.

2.2 Serverless-arkkitehtuurin haitat

Serverless-arkkitehtuuriin liittyy myös jonkin verran rajoituksia ja heikkouksia, mikä tulee ottaa huomioon, jos on siirtämässä nykyistä sovellusta serverless-alustalle, tai on kehittämässä uutta projektia serverless-ympäristöön.

2.2.1 Kylmäkäynnistykset

Coldstartit eli kylmäkäynnistykset ovat yksi serverless-arkkitehtuurin rajoitteista. Kylmäkäynnistyksellä tarkoitetaan tilannetta, jossa funktiota ei olla suoritettu vähään aikaan, jonka seurauksena funktio menee pois päältä. Esimerkiksi yksittäinen Azure Functions funktio pysyy päällä maksimissaan 10 minuuttia (.NET Handbook, 2023). Tämän jälkeen funktio menee niin sanottuun kylmään tilaan.



Kuva 3. Kylmäkäynnistyksen tyypillinen kesto (Thundra, 2019)

Kun heräte käynnistää funktion, joka on kylmässä tilassa, kestää funktion käynnistämisessä normaalia kauemmin, sillä funktiosta luodaan uusi instanssi, mikä aiheuttaa normaalia pidemmän viiveen funktion suorittamiselle. Joissakin tapauksissa kylmäkäynnistyksen viiveet voivat olla jopa useita sekunteja kuten kuvasta 3. voidaan havaita. Kylmäkäynnistysten aikaa on toki mahdollista lyhentää, esimerkiksi allokoimalla funktiolle lisää muistia (Rehemägi, 2021).

2.2.2 Vendor lock-in

Serverless-arkkitehtuurissa vendor lock-in-ongelma on yleistä, koska useimmat serverless-palvelut tarjoavat erittäin rajoitetun hallinnan palvelimen ympäristöstä ja sovellusarkkitehtuurista. Tämä tarkoittaa, että palvelun käyttäjä on riippuvainen toimittajan tarjoamasta infrastruktuurista ja palveluista (Cloudflare, n.d.).

Myös sovelluksen riippuvuudet ja integraatiot toimittajan tarjoamiin palveluihin ovat yleensä tiukasti sidoksissa toisiinsa, mikä tekee sovelluksen siirtämisen toiseen palveluun vaikeaksi. Tämä voi aiheuttaa ongelmia, jos sovellus halutaan viedä toisen palveluntarjoajan alustalle (Cloudflare n.d.).

2.2.3 Rajoitettu suoritusaika

Yksi serverless-ympäristön rajoitteista on myös se, että funktiota ei suoriteta ikuisesti. Palveluntarjoajilla on omat rajoitteensa sen suhteen, kuinka kauan yksittäistä funktioita voidaan suorittaa. Esimerkiksi AWS Lambdassa funktion maksimi suoritusaika on 15 minuuttia (Tripathy, 2022).

Jos funktio suorittaa vaativia tai raskaita prosesseja, sen suoritusaika voi kasvaa suuremmaksi kuin sallittu ja tämä johtaa keskeytykseen ennen kuin koodi tai prosessi on kokonaan suoritettu. Tästä syystä on tärkeää huomioida, että funktion suoritus voi katketa, vaikka koodi tai prosessi olisi vielä kesken.

3 C# JA .NET CORE

C# on Microsoftin kehittämä ja julkaisema yleiskäyttöinen korkeantason oliopohjainen ohjelmointikieli. C# on suhteellisen uusi ohjelmointikieli, sillä sen kehittäminen alkoi Microsoftilla työskennelleen Anders Hejlsbergin johdolla 1990-luvun lopulla. Projektin alkuvaiheessa C# ohjelmointikieltä nimitettiin Cooliksi. C# tavoitteena oli luoda moderni, yleiskäyttöinen ja oliopohjainen ohjelmointikieli, joka perustuu C++ ja Java ohjelmointikieliin (DotNetsExpert, 2023) Ensimmäinen virallinen versio C#-kielestä julkaistiin vuonna 2002 .NET 1.0 julkaisun yhteydessä.

C# muistuttaakin syntaksiltaan hyvin paljon C++ ja Java ohjelmointikieliä. C#-kielessä on paljon hyviä ominaisuuksia, kuten automaattinen roskienkeruu ja staattinen tyypitys. Automaattisen roskienkeruun ansiosta muistin allokoinnista ja vapauttamisesta ei tarvitse erikseen huolehtia, vaan roskienkeruu hoitaa tämän automaattisesti (Microsoft, 2023). Staattisessa tyypityksessä tietotyypit tarkistetaan ennen ohjelman suorittamista, jolloin esimerkiksi kääntäjä antaa virheen, jos muuttujalla on väärä tietotyyppi. Tämä mahdollistaa virheiden havaitsemisen jo ennen ohjelman suorittamista. Koodista tulee myös luettavampaa ja helpompaa ylläpitää, koska jokaisen muuttujan tietotyyppi on erikseen määritelty (BairesDevBlog n.d.).

.NET Core on Microsoftin kehittämä avoimen lähdekoodin cross-platform alusta, jolla voidaan kehittää sovelluksia Windows, Linux, MacOs, pilvi- ja mobiili ympäristöihin. .NET Corella voidaan toteuttaa esimerkiksi ohjelmointirajapintoja, työpyötä- ja mobiilisovelluksia, pelejä, Iot-sovelluksia ja pilvinatiiveja sovelluksia. .NET Core tukee C#, F# ja Visual Basic ohjelmointikieliä, vaikkakin C# on kuitenkin .NET Core alustan suosituin ohjelmointikieli (Microsoft n.d.).

4 ESIMERKKISOVELLUS

Esimerkkisovellus on yksinkertainen REST-API, joka toimii lokaalissa kehitysympäristössä, ja valmiit serverless-implementoinnit tullaan viemään palveluntarjoajien tuotantoympäristöihin. API:lla voidaan luoda, päivittää, lukea ja poistaa äänestyksiä (Poll). Äänestyksien runko pitää sisällään otsikon, sekä äänestysvaihtoehdot. Esimerkiksi äänestystä luodessa nimetään äänestys ja sille annetaan vaihtoehtoja, ja taas äänestettäessä valitaan jokin äänestyksessä oleva vaihtoehto.

Tässä tapauksessa API käyttää näiden toimintojen toteuttamiseen CRUD operaatioita. CRUD on lyhenne neljästä perustoiminnosta, joita yleensä käytetään API:n tietokantatapahtumissa. Create(luo), Read(lukee), Update(päivittää) ja Delete(poistaa) resursseja. API-kerroksessa on määritelty CRUD operaatiota vastaavia HTTP-metodeita kuten, GET, POST, PUT ja DELETE.

API on toteutettu .NET 6 - kehyksellä ja se hyödyntää Entity Framework Corea (EF Core) tietokantatoiminnallisuuksien hallintaan. EF Core on ORM, jonka avulla tietokantataulut voidaan mallintaa olioksi ohjelmakoodissa, lisäksi EF Core mahdollistaa tietokantakyselyiden suorittamisen suoraan .NET Coren LINQ-kyselyiden avulla, joka mahdollistaa tietokantakyselyiden tekemisen suoraan C#-kielellä. Sovelluksen sisäisessä tiedonsiirrossa käytetään AutoMapper-työkalua. AutoMapperin avulla voidaan muuntaa esimerkiksi tietokantaolio eri DTO-olioksi ja toisinpäin.

Lisäksi EF Coren avulla voidaan toteuttaa tietokantamigraatiot sekä määrittää käytettävä tietokanta (Microsoft, 2023e). Kehitysympäristössä käytetään tietokantana Microsoft SQL Server tietokantaa, ja tuotantoympäristössä käytetään Azure SQL tietokantaa.

4.1 Sovelluksen rakenne

Sovellus on toteutettu kerrosarkkitehtuurilla. Kerrosarkkitehtuuri parantaa koodin ylläpidettävyyttä ja testattavuutta, helpottaa riippuvuuksien hallintaa, sekä tekee koodista uudelleenkäytettävää (Microsoft, 2023a).

Kerrosarkkitehtuuri tarkoittaa sitä, että sovelluksen keskeiset toiminnot ovat jaettu omiin kerroksiinsa taulukon 1. mukaisesti. Jokaisen kerroksen on tarkoitus vastata tietystä osa-alueesta sovelluksen toiminnan kannalta, kuten tietokannasta, business-logiikasta ja API toiminnoista. Sovelluksen toiminnot ovat eristetty neljään keskeiseen kerrokseen: DataAccess, LogicInterface, Logic ja PollApi.

Taulukko 1. Sovelluksen kerrokset.

Kerros	Kuvaus
DataAccess	DataAccess-kerros vastaa tietokantaan liittyvistä toiminnoista. Tässä kerroksessa määritellään Entity Frameworkilla tietokantaentiteetit, sekä tietokantarelaatiot.
LogicInterface	LogicInterface-kerros sisältää rajapinnat, joita käytetään API:n business-logiikassa. Rajapinnat määrittelevät, mitä toimintoja äänestyksiin liittyen voidaan suorittaa. Lisäksi tässä kerroksessa määritellään DTO-luokat, joita käytetään tietojen siirtämiseen eri kerrosten välillä.
Logic	Logic-kerros on vastuussa API:n business-logiikan suorittamisesta. Tässä kerroksessa implementoidaan rajapintojen määrittelemät toiminnot, ja injektoidaan DataAccess-kerroksen tietokantaentiteetit- ja operaatiot. Varsinaiset CRUD-operaatiot suoritetaan tässä kerroksessa.
PollApi	PollApi-kerros on rajapintakerros, joka vastaanottaa ja käsittelee HTTP-pyyntöjä. PollApi kerroksessa oleva kontrolleri vastaa eri polkujen kautta tapahtuviin HTTP-pyyntöihin.

PollApi kerroksessa oleva kontrolleri (PollController.cs) vastaanottaa HTTP-kutsuja, ja palauttaa HTTP-vastauksen käyttäjälle. Kontrollerissa oleville päätepisteille on määritelty polut taulukossa 2., joilla päätepisteissä olevia metodeita kutsutaan.

Taulukko 2. Kontrollerin päätepisteet

HTTP-pyyntö	Polku	Toiminto
GET	/Polls	Hakee kaikki olemassa olevat äänestykset
GET	/Polls/{id}	Hakee yksittäisen äänestyksen
PUT	/Polls/{id}/vote/{optionId}	Äänestää valittua vaihtoehtoa
DELETE	/Polls/{id}	Poistaa yksittäisen äänestyksen
POST	/Polls	Luo uuden äänestyksen

```
[HttpGet]
public async Task<ActionResult<ICollection<Poll>>> Get()
{
    var result = await pollService.GetPollsAsync();
    return Ok(result);
}
```

Koodi 1. Get-päätepiste

Get-päätepiste (koodi 1.) mahdollistaa äänestysten hakemisen tietokannasta. Get-päätepiste kutsuu logiikkakerroksen PollService luokassa toteutettua GetPollsAsync() metodia. GetPollsAsync() metodi hakee listana kaikki äänestykset tietokannasta. Kun äänestykset ovat haettu, ne tallennetaan result- nimiseen muuttujaan. Lopuksi Ok-metodi palauttaa vastauksena HTTP-statuskoodin 200, sekä result-muuttujan sisältämän datan JSON-muodossa.

5 .NET API:N MIGRATOINTI SERVERLESS-YMPÄRISTÖIHIN

Kuten johdannossa mainittiin, työn päätarkoituksena on tutkia miten olemassa oleva .NET Core:lla toteutettu REST-API saadaan migratoitua Azure Functions, Amazon Lambda ja Google Cloud Functions ympäristöihin. Serverless-arkkitehtuuri on osoittautunut houkuttelevaksi vaihtoehdoksi, koska se tarjoaa monia etuja, kuten kustannustehokkuutta, automaattista skaalautuvuutta sekä joustavuutta kehitystyössä (Karim Hafez n.d.).

Tässä luvussa käydään läpi käytännön vaiheet .NET API:n migraatiosta perinteisestä palvelinympäristöstä serverless-ympäristöön. Tulemme käymään läpi sovelluksen koodilliset muutokset, sekä muut määrittelyt, joita tarvitsee tehdä ympäristön vaihdon yhteydessä. Näitä muutoksia ovat esimerkiksi riippuvuusinjektion määrittely ja serverless-konfiguraatioiden mukauttaminen jokaisen palveluntarjoajan ympäristöille.

Kaikille funktiolle määritellään herätteeksi HTTP-heräte, jotta sovellus saadaan toimimiaan API:n tavoin myös serverless-ympäristössä. Lisäksi tässä kappaleessa perehdytään siihen, miten valmiit servess-funktioksi muutetut toiminnot viedään serverless-ympäristöön.

Kun tarvittavat muutokset ja koodin refaktoroinnit .NET API:ssa on tehty, voidaan aloittaa varsinainen siirtoprosessi edellä mainittuihin serverless-ympäristöihin. Siirtoprosessissa viedään refaktoroidut versiot REST-API:sta AWS Lambda, Azure Functions ja Google Cloud Functions serverless-ympäristöihin. Tämä prosessi sisältää valmiin serverless-sovelluksen paketoinnin, sekä lataamisen palveluntarjoajien ympäristöihin. Lisäksi kappaleessa käydään läpi jokaisen palveluntarjoajan työkalut, joiden avulla varsinainen migraatio palveluntarjoajan ympäristöihin saadaan suoritettua.

5.1 Azure Functions

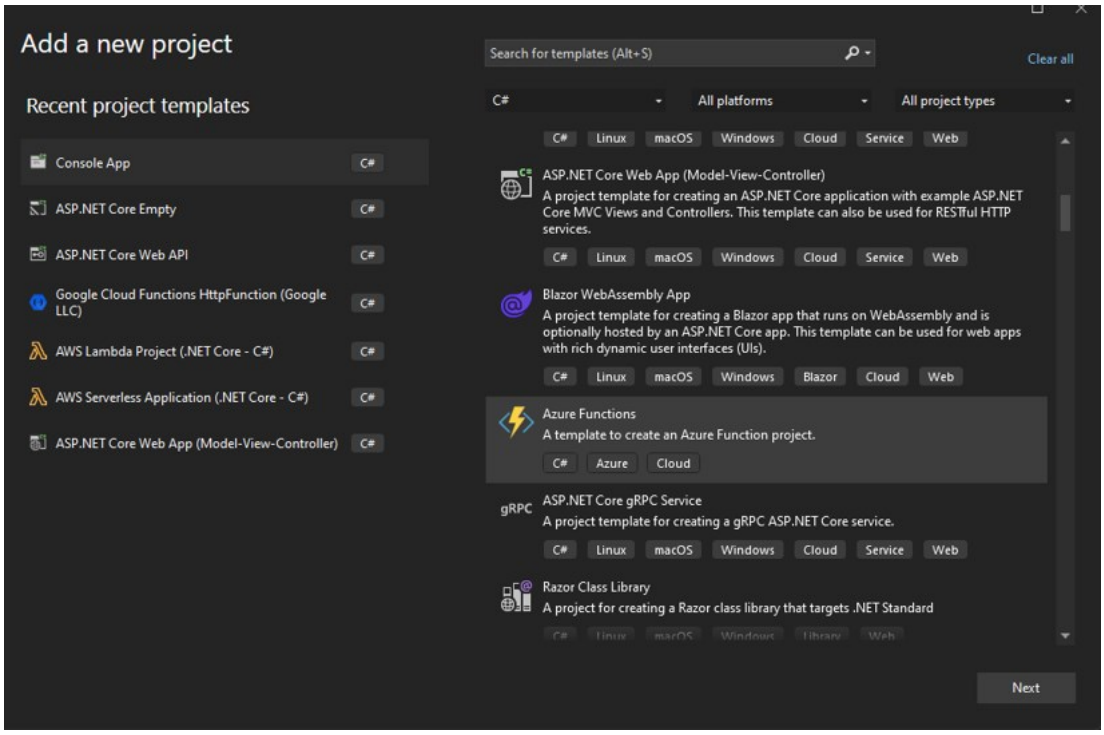
Azure Functions julkaistiin vuonna 2016 Microsoftin Azure pilvipalvelussa. Koska Azure Functions on Microsoftin palvelu, siinä on laajempi tuki Microsoftin omille teknologioille, kuten TypeScriptille, C# ohjelmintikielelle, sekä .NET-ohjelmistokehykselle (Microsoft, 2023c).

Azure Functions-funktioita voidaan kehittää suoraan Visual Studiossa. Serverless-framework tukee myös sovelluskehitystä Azure Functions ympäristön kanssa. Azure tarjoaa myös mahdollisuuden kehittää serverless-sovelluksia suoraan webkäyttöliittymässä olevan koodieditorin avulla.

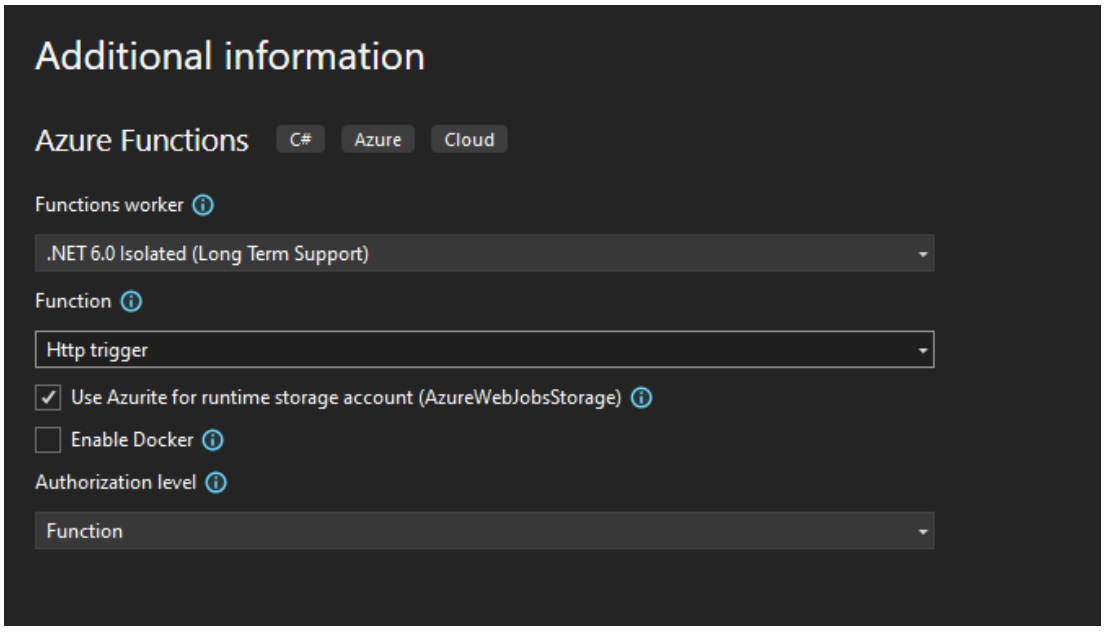
5.1.1 Koodimuutokset

Ensimmäisenä nykyiseen projektiin lisätään uusi Azure Functions- projekti. Visual Studio tarjoaa valmiin projektipohjan Azure Functions projektille ja tässä toteutuksessa käytetään kuvassa 4. olevaa pohjaa. Tämän jälkeen Funktion suoritusympäristöksi valitaan .NET 6 Isolated kuvan 5. mukaisesti.

.NET 6 Isolated ympäristössä on erona normaaliin .NET 6 ympäristöön se, että esimerkiksi riippuvuusinjektio on helpompi toteuttaa ja väliohjelmistoputken (Middleware-pipelinen) käyttäminen on mahdollista (Microsoft, 2023b). Väliohjelmistolla voidaan esimerkiksi käsitellä HTTP-pyyntöjä kontrollerin ulkopuolella.



Kuva 4. Azure Functions projektipohjan lisääminen sovellukseen



Kuva 5. Azure Functions projektin määrittely

Kun Azure Functions projektipohjasta on luotu projekti, Azure Functions- projektin hakemistoon on muodostettu tarvittavat tiedostot funktioiden luomista varten.

Program.cs- tiedostossa konfiguroidaan sovelluksen riippuvuusinjektio ja asetuksia sovelluksen käynnistymisen kannalta, samalla tavalla kuten normaalissa .NET ympäristössä.

Tähän tiedostoon voidaan määrittää esimerkiksi sovelluksen käyttämät riippuvuudet, tietokantayhteydet sekä mahdolliset muut toiminnot, kuten väliohjelmistot. Eli nykyisen API:n käyttämät riippuvuudet, kuten AutoMapper, logiikkakerrokset ja tietokantakonteksti saadaan määriteltyä kuvan 6. mukaisesti tässä tiedostossa.

```
var host = new HostBuilder()
    .ConfigureFunctionsWorkerDefaults()
    .ConfigureServices(s =>
    {
        var connectionString = Environment.GetEnvironmentVariable("DbConnection", EnvironmentVariableTarget.Process);
        s.AddScoped<IPollService,PollService>();
        s.AddDbContext<DataContext>(options =>
        {

            options.UseSqlServer(connectionString!);
        });
        s.AddAutoMapper(typeof(MappingProfile));


    })
    .Build();

await host.RunAsync();
```

Kuva 6. Program.cs luokan ConfigureServices-metodin määrittely

Koska API:a ollaan migratoimassa Azure Functions ympäristöön, alkuperäinen API-kerros on poistettu, ja se on korvattu Azure Function projektilla. Funktiot määritellään omaan tiedostoonsa. Projektipohjan tekemä Functions1.cs tiedosto pitää sisällään yhden esimerkki funktion. Tässä tapauksessa nimetään Functions1.cs-tiedosto paremmin käyttötarkoitusta kuvaavammaksi (PollFunctions.cs) ja tähän tiedostoon tullaan ohjelmoimaan kaikki tarvittavat funktiot.

PollFunctions.cs-tiedosto korvaa PollApi kerroksessa käytetyn kontrollerin. Koska sovellus on toteutettu kerrosarkkitehtuurilla, voidaan funktioille tuoda

tarvittavat riippuvuudet riippuvuusinjektion avulla, kuten logiikkakerroksen metodit äänestysten käsittelylle.

Esimerkiksi Get-endpoint, joka löytyy pohjatoteutuksen kontrollerista ja toteutetaan funktiona kuvan 7. mukaisesti.

```
[Function("Get")]
public async Task<HttpResponseData> Get([HttpTrigger(AuthorizationLevel.Function, "get", Route = "polls")] HttpRequestData req)
{

    var polls = await pollService.GetPollsAsync();
    var response = req.CreateResponse(HttpStatusCode.OK);
    await response.WriteAsJsonAsync(polls);

    return response;
}
```

Kuva 7. Azure Funktion määrittely

Funktiolle määritetään valtuutustasoksi function, mikä tarkoittaa sitä, että funktiota ei voida kutsua ilman funktioavainta (Function access key). Funktioavain on avain, jolla Azure-ympäristö varmistaa, että funktion kutsujalla on valtuudet suorittaa funktio. Jos funktiota yrittää kutsua ilman avainta tai väärällä avaimella, palauttaa kutsu HTTP-statuskoodin 401 joka indikoi, että funktion kutsujalla ei ole valtuuksia suorittaa kyseistä funktiota. Jos valtuutustason määrittelee anonyymiksi, voidaan funktiota kutsua ilman funktioavainta.

Herätteen tyypiksi määritetään HTTP-kutsu, ja HTTP-metodiksi määritellään Get, sekä reitiksi annetaan ”/polls”. Sitten funktio kutsuu logiikkakerroksesta GetPollsAsync()-metodia, joka hakee kaikki äänestykset ja palauttaa vastauksen JSON-muodossa, loput funktiot määritellään vastaavalla tavalla. Eli funktioille määritetään valtuutustaso, heräte, polku, HTTP-metodi sekä data, jonka funktio palauttaa.

5.1.2 API:n siirtäminen Azure Functions ympäristöön

Azure Functions projektin vieminen Azureen onnistuu usealla eri tavalla, esimerkiksi CI/CD pipelinen, Visual Studion, Azure Pipelinesin, GitHub Actionien, serverless frameworkin tai Azure Core Toolsin kautta. Tässä toteutuksessa käytämme Visual Studiota sovelluksen viemiseen Azuren ympäristöön, joten ulkoisia työkaluja, kuten Azure Core Toolsia, ei tarvita.

Kun Azuressa on määritelty sovellukselle resurssiryhmä ja funktioinstanssit, voidaan sovellus julkaista suoraan Visual Studion kautta. Visual Studiossa valitaan vain resurssiryhmä, johon sovellus julkaistaan. Visual Studion työkalu suorittaa sovelluksen pakkaamisen sekä viemisen Azuren ympäristöön.

Kun sovellus on julkaistu Azure-ympäristöön, ovat funktiot tulleet resurssiryhmän funktio instansseihin näkyiin kuvan 8. mukaisesti.

Name	Trigger	Status	Monitor
Create	HTTP	Enabled	Invocations and more
Delete	HTTP	Enabled	Invocations and more
Get	HTTP	Enabled	Invocations and more
GetById	HTTP	Enabled	Invocations and more
Vote	HTTP	Enabled	Invocations and more

Kuva 8. Funktiot Azure Portal näkymässä.

Jokaiselle funktiolle on luotu oma URL, jonka kautta heräte suorittaa funktion. Esimerkiksi Get-funktiolle on luotu URL seuraavasti: https://projekti.azurewebsites.net/api/polls?code=funktioavain==. Koska funktiolle on asetettu valtuustasoksi funktioavain, tulee code-parametrille antaa funktioavain aina kutsua tehdessä. Funktioavaimet generoituivat Azure-ympäristöön automaattisesti.

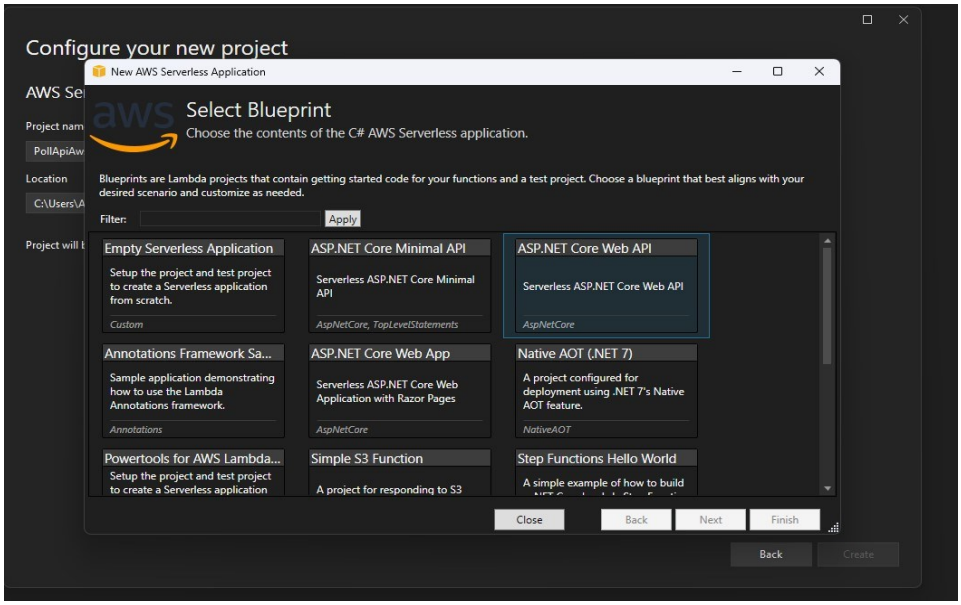
5.2 AWS Lambda

AWS Lambda on Amazon Web Services pilvipalvelun tarjoama serverless-alusta. AWS Lambda julkaistiin vuonna 2014, ja se toi serverless-palvelut suureen suosioon (TIVI, 2018). AWS Lambdassa on hyvä tuki yleisille ohjelmointikielille. AWS Lambdaan voidaan kehittää sovelluksia serverless-frameworkin avulla, tai AWS-palveluiden omalla AWS-SAM-työkalulla tai AWS-alustan omalla webkäyttöliittymän koodieditorilla.

5.2.1 Koodimuutokset

Kuten Azure Functions tapauksessa, tästä projektista tullaan poistamaan alkuperäinen API-kerros, ja se korvataan AWS Lambda projektilla. AWS tarjoaa Visual Studioon omat projektipohjat Lambda funktioiden kehittämistä varten (kuva 9.).

AWS Lambda funktioiden viemiseen AWS ympäristöön on tarjolla useita työkaluja, ja tässä ratkaisussa päädytään AWS-SAM: iin. AWS-SAM on työkalu, jonka avulla valmis AWS Lambda sovellus voidaan viedä suoraan palveluntarjoajan ympäristöön hyödyntäen AWS Lambda projektipohjan mukana tullutta template.yaml tiedostoa.



Kuva 9. AWS Lambda projektipohjan valinta Visual Studiossa

Projektipohja generoi AWS Lambda projektia varten tarvittavat tiedostot. Näistä tärkeimmät ovat: Startup.cs, serverless-template.yaml sekä LamdaEntryPoint.cs. Startup.cs-tiedostoon konfiguroidaan Lambda sovelluksen riippuvuudet, sekä sen käyttämät palvelut. Konfigurointi tapahtuu lähes vastaavasti kuin Azure Functions toteutuksen Progam.cs tiedostossa (kuva 6.).

```
public void ConfigureServices(IServiceCollection services)
{
    var connectionString = Environment.GetEnvironmentVariable("DbConnection")
    services.AddControllers();
    services.AddScoped<IPollService, PollService>();
    services.AddDbContext<DataContext>(opt =>
    {
        opt.UseSqlServer(connectionString);
    });
    services.AddAutoMapper(typeof(MappingProfile));
}
```

Kuva 10. Startup.cs-tiedoston ConfigureServices-metodin määrittely AWS Lambda projektissa

LambdaEntryPoint.cs-luokka on varsinainen funktioiden aloituspiste. Kaikki sovelluksessa määritellyt funktiot käynnistyvät kyseisen luokan kautta. LambdaEntryPoint.cs luokka perii Amazon.Lambda.AspNetCoreServer nimiavaruuden APIGatewayProxFunction luokan, joka on AWS:n tarjoama laajennus .NET ympäristöön, mikä mahdollistaa .NET sovelluksen suorittamisen AWS Lambdan ajoympäristössä.

LamdaEntryPoint.cs suorittaa Startup.cs-tiedoston, jossa .NET sovelluksen tarvitsemat konfiguraatiot ovat määritelty, kuten riippuvuusinjektio ja tietokantakerros (kuva 10.). Koska Lambda sovellus ajetaan LambdaEntrypointin kautta, voidaan Lambda toteutuksessa käyttää alkuperäistä kontrolleria, joka sisältää kaikki päätepisteet HTTP-kutsuille. Näin ollen kontrolleri voidaan toteutettua alkuperäisen toteutuksen mukaisesti (koodi 1.), eikä se tarvitse erikseen uusia määrittelyjä tai koodimuutoksia. Tämän ansiosta alkuperäisen

kontrollerin koodia voidaan käyttää suoraan sellaisenaan AWS Lambda sovelluksessa.

Varsinainen serverless-konfigurointi AWS Lambda ympäristöön tehdään omaan template.yaml tiedostoon, jossa määritetään funktiolle alkupiste ja funktion käsittelijä, infrastruktuuri kuten allokoitu muisti, ajoympäristö, herätteen tyyppi, käytettävä HTTP-metodi, sekä Amazon Api Gatewaylle reitti funktion herätettä varten.

```
DeletePollFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: PollApiAws::PollApiAws.LambdaEntryPoint::FunctionHandlerAsync
    Runtime: dotnet6
    CodeUri: ''
    MemorySize: 256
    Timeout: 30
    Role:
    Policies:
    - AWSLambda_FullAccess
    Events:
      DeletePollFunctionEvent:
        Type: Api
        Properties:
          Path: "/Polls/{id}"
          Method: DELETE
```

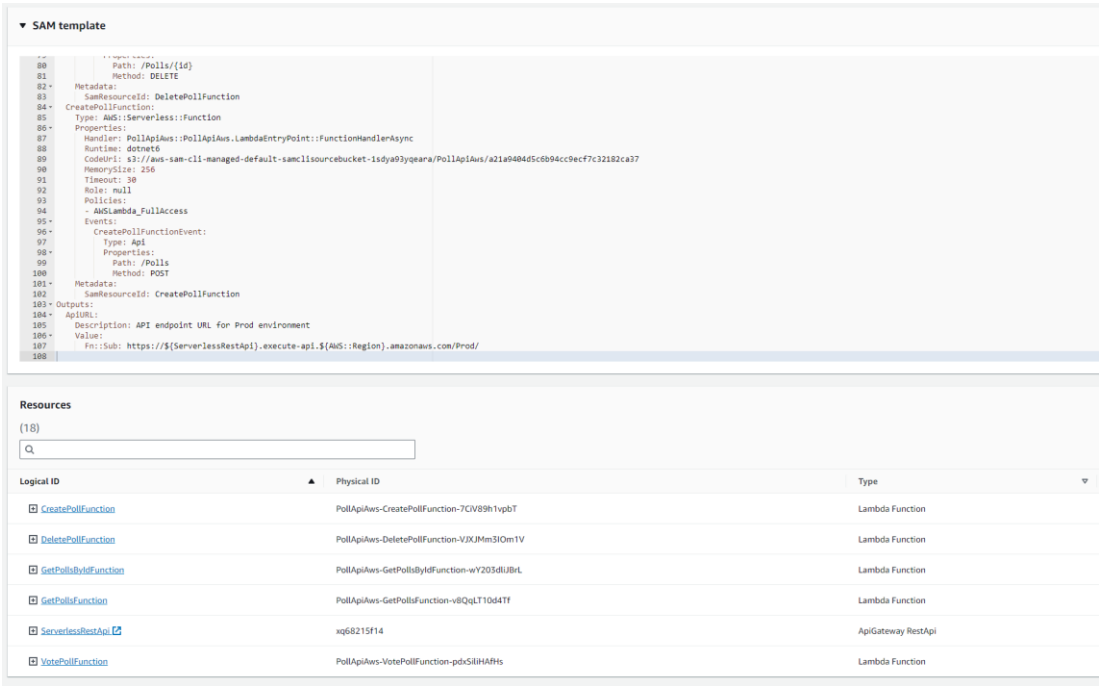
Kuva 11. Funktion määrittely template.yaml tiedostossa.

Kaikki funktiot ja niiden infrastruktuuri määritellään template.yaml tiedostoon vastaavalla tavalla kuten kuvassa 11. ja template.yaml toimii ikään kuin pohjapiirustuksena AWS serverless-ympäristöä varten.

5.2.2 API:n siirtäminen AWS Lambda ympäristöön

AWS Lambda ympäristöön valmis projekti voidaan viedä zip-tiedostossa, Amazonin AWS CodePipelinella, jonka avulla on myös mahdollista tehdä CI/CD pipeline. Vienti on myös mahdollista toteuttaa GitHub Actionsin, serverless frameworkin tai AWS SAM:in avulla. Projektissa luodun template.yaml tiedoston ansiosta, vienti AWS Lambda ympäristöön onnistuu AWS SAM:in avulla helposti. AWS SAM käyttää sovelluksen pakkaamiseen ja julkaisemiseen template.yaml tiedostoa. `sam build` komennolla AWS SAM asentaa projektin riippuvuudet ja pakkaa sovelluksen AWS Lambda ympäristöön toimivaksi paketiksi.

`sam deploy` komento julkaisee projektin AWS Lambda serverless-ympäristöön, luo resurssiryhmän Lambda projektille AWS Consoleen, sekä määrittelee AWS API Gatewayn projektiin. API Gateway hyödyntää template.yaml tiedostossa määriteltyjä polkuja ja HTTP-metodeita, jonka avulla Lambda funktiolle voidaan tehdä HTTP-kutsuja REST API:n tavoin (Amazon n.d.).



Kuva 12. AWS Lambda funktiot ja API Gateway AWS Consolessa

`sam deploy` komennon jälkeen template.yaml tiedostossa määritellyt Lambda funktiot ovat luotu AWS Consoleen, lopputulos näkyy kuvassa 12. AWS SAM on tehnyt myös template.yaml-tiedoston avulla API Gatewayn, jonka kautta funktiota voidaan kutsua.

5.3 Google Cloud Functions

Google Cloud Functions julkaistiin vuonna 2018 Google Cloud -pilvipalveluun. Google Cloud Functions tarjoaa hyvän tuen suosituille ohjelmointikielille, aivan kuten edellä mainitut ympäristöt (Google n.d.). Sovellusten kehittäminen Google Cloud Functions ympäristöön onnistuu Google Cloud CLI-komentoliittymällä, tai Google Cloud Consolen webkäyttöliittymän avulla.

5.3.1 Koodimuutokset

Kuten edellisissä tapauksissa, myös Google Cloud Functions alustalle löytyy valmis projektipohja, jonka avulla .NET ympäristössä toimivia funktioita voidaan toteuttaa. Projektipohja tarjoaa myös tarvittavat laajennukset .NET ajoympäristölle, kuten Functions Framework for .NET ohjelmistokehyksen, jonka avulla C# koodia voidaan suorittaa Google Cloud Functions ympäristössä.

Riippuvuusinjektio ja sovelluksen muu konfiguraatio määriteltyä Startup.cs-tiedostossa. Startup.cs perii Google.Cloud.Functions.Hosting nimiavaruuden FunctionsStartup.cs luokan. Itse Startup.cs-tiedosto määritellään vastaavalla tavalla kuin AWS Lambdassa (kuva 10.). Eli tarvittavat riippuvuudet ja palvelut lisätään ConfigureServices-metodissa.

Itse funktion määrittely eroaa huomattavasti verrattuna AWS Lambdaan tai Azure Functionsiin. Jokainen funktio määritellään omaan luokkaansa sen

sijaan, että ne voitaisiin tehdä yhteen paikkaan kuten AWS Lambdassa tai Azure Functionssissa.

```
namespace PollApiGCP.PollFunctions
{
    [FunctionsStartup(typeof(Startup))]
    public class CreatePollFunction : IHttpFunction
    {
        private readonly IPollService pollService;
        public CreatePollFunction(IPollService pollService)
        {
            this.pollService = pollService;
        }

        public async Task HandleAsync(HttpContext context)
        {
            try
            {

                JsonSerializerOptions jsonSerializerOptions = new JsonSerializerOptions()
                {
                    PropertyNameCaseInsensitive = true,
                };

                var addPollRequestDto = await JsonSerializer.DeserializeAsync<AddPollRequestDto>(context.Request.Body, jsonSerializerOptions);
                var createdPoll = await pollService.CreatePollAsync(addPollRequestDto);
                await context.Response.WriteAsJsonAsync(createdPoll);
            }
            catch (JsonException)
            {
                context.Response.StatusCode = (int)HttpStatusCode.BadRequest;
                await context.Response.WriteAsync("Invalid JSON data in the request body.");
                return;
            }
        }
    }
}
```

Kuva 13. Google Cloud Functions määrittely CreatePollFunction luokassa

Google Cloud Functions tarjoaa IHttpFunction–rajapinnan HTTP-pyyntöjen käsittelyyn. Kuten ratkaisusta näkee (kuva 13.), IHttpFunction rajapinnan HandleAsync(HttpContext context) metodi tarjoaa vain HttpContext-luokan, josta saadaan HTTP-pyyntöihin liittyvät tiedot kuten, reitti, query-parametrit, HTTP-pyyntö sekä HTTP-vastaus. Reititystä ei voi validoida suoraan, mikä puolestaan on mahdollista Azure Functions-funktioissa tai MVC kontrollerissa, joita AWS Lambda projekti ja alkuperäinen API-kerros käyttävät.

Google Cloud Functions-funktioden logiikka vaatii enemmän käsittelyä, varsinkin tilanteessa, jossa funktio käyttää HTTP-herättettä. Esimerkiksi jos HTTP-pyynnöstä halutaan saada vaikkapa äänestyksen yksilöivä Id talteen HTTP-pyynnön polusta, Id:n arvo tulee ottaa talteen HttpContext-luokasta manuaalisesti. Tämä hankaloittaa funktioiden toiminallisuuksien tekemistä varsinkin, jos funktioiden on tarve useita eri päätepisteitä, jotka vastaanottavat parametrejä polkujen kautta.

5.3.2 API:n siirtäminen Google Cloud Functions ympäristöön

Kun Google Cloud Consoleen on luotu Google Cloud Functions projekti, voidaan funktiot viedä palveluntarjoajan ympäristöön. Vienti onnistuu CI/CD pipelinen, Google Cloud CLI:n tai serverless frameworkin avulla. Tässä toteutuksessa projekti viedään Googlen Cloudin serverless ympäristöön Google Cloud CLI:n avulla. AWS Lambda projektissa ja Azure Functions projektissa on mahdollista viedä kaikki funktiot yhdellä komennolla, kun Google Cloud CLI:ssä voidaan viedä yksi funktio per komento.

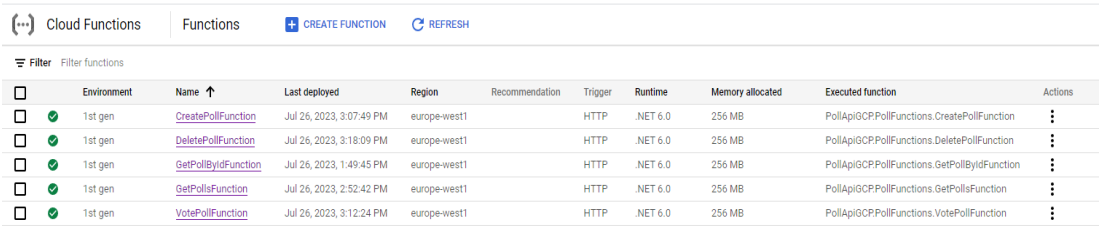
Funktiot viedään Google Cloud Functions ympäristöön `gcloud functions deploy`-komennolla. Komentoon määritellään myös muita tarvittavia parametreja, kuten Google Cloud Consolessa olevan projektin ID, funktion ajoympäristö, funktion herätetyyppi ja funktion suorituspiste. Esimerkiksi funktio, joka hakee äänestykset yksilöidyn id:n perusteella, vietäisiin Google Cloud Functions ympäristöön seuraavalla komennolla:

```
gcloud functions deploy
--runtime dotnet6
--trigger-http
--entry-point PollFunctions.GetPollByIdFunction
--allow- unauthenticated
--project<projektin-id>
```

Koodi 2. Komento funktion viemiselle Google Cloud ympäristöön

Google Cloud CLI-komento koodin 2. mukaisesti pakkaa projektin ja siirtää sen Google Cloud Functions ympäristöön, jonka jälkeen se luo tai päivittää funktion Google Cloud Consolessa olevaan projektiin, sekä pystyttää sen määriteltyyn ajoympäristöön (runtime), joka on määritelty komennossa (koodi 2.). Tämä generoi funktiolle URL:in, jonka avulla funktiota voidaan kutsua. Kun

prosessi on valmis, funktio on käytettävissä Google Cloud Functions ympäristössä (kuva 14.), ja funktio on kutsuttavissa.



Kuva 14. Valmiit funktiot Google Cloud Consolessa

6 MIGRAATION TULOKSET

Alkuperäinen REST-API oli suunniteltu ja toteutettu perinteisessä palvelin ympäristössä. Kun API:t ovat migratoitu Google Cloud Functions, AWS Lambda ja Azure Functions ympäristöihin, on aika tarkastella API:n toimintaa ja verrata sen toimintaa alkuperäisen toteutuksen kanssa. Migraation myötä API:n business-logiikka on säilynyt ennallaan, mutta jokaisen serverless-ympäristöön migratoidun API:n taustalla oleva infrastruktuuri muuttunut merkittävästi.

Infrastruktuurimuutosten myötä tulemme tarkastamaan, että serverless-ympäristöihin viedyt API:t toimivat ja käyttäytyvät samalla tavalla kuin ennen migraatiota, niin business-logiikan, HTTP-kutsujen kuin riippuvuuksienkin osalta. Myös migraation myötä tietokanta vaihtui Microsoft SQL Server tietokannasta Azuressa olevaan SQL Server tietokantaan.

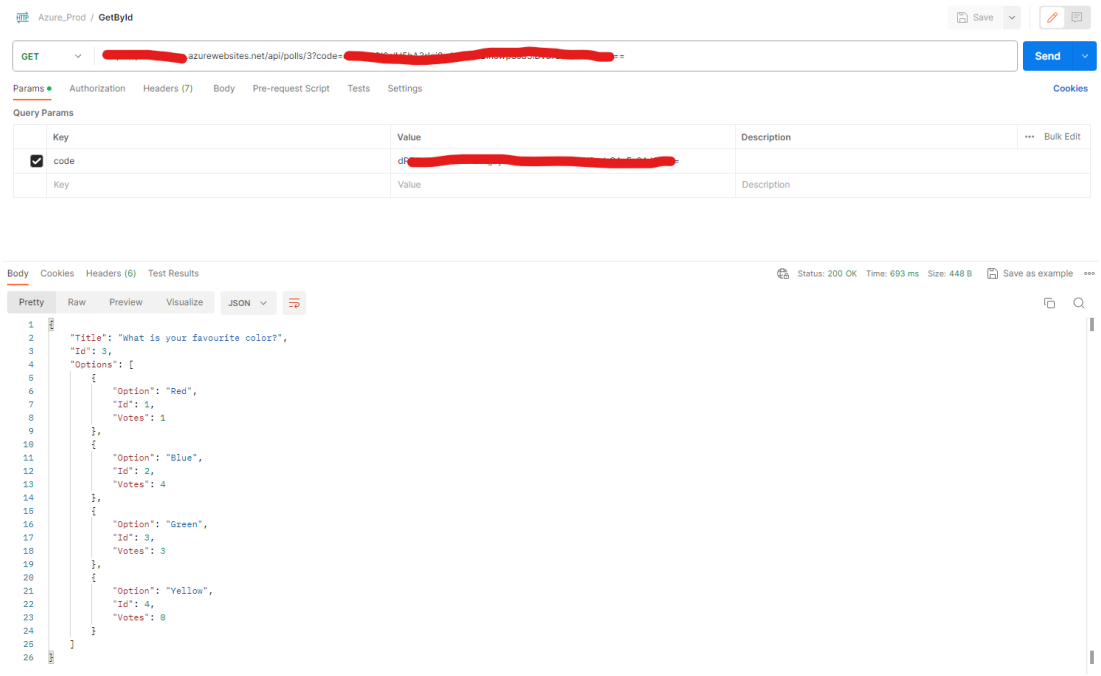
Migratoitujen API:en testaamiseen käytetään Postman nimistä REST-API testaustyökalua. Postmanin avulla voidaan lähettää HTTP-kutsuja rajapinnoille, sekä nähdä mitä rajapinta palauttaa, kuten HTTP-statuskoodin, sekä kutsun onnistuessa myös rajapinnan palauttaman datan. Tämän avulla voidaan varmistua siitä, että rajapinta käyttäytyy ja toimii toivotulla tavalla.

Postmanin avulla tullaan testaamaan jokainen serverless-ympäristöihin viety serverless-sovellus. HTTP-kutsut tehdään URL: ien kautta, jotka generoituivat migraation yhteydessä palveluntarjoajien ympäristöihin.

AWS Lambda, Google Cloud Functions ja Azure Functions tarjoavat kaikki omat työkalunsa pilvi- ja serverless-sovellusten monitorointiin, joiden avulla mahdollisia virhetilanteita voidaan selvittää ja tutkia.

Taulukko 3. Testauksen tulokset Azure Functions ympäristössä

HTTP-pyyntö	Polku	Toimii odo-tetusti	Toimii osit-tain	Ei toimi
GET	/Polls	x		
GET	/Polls/{id}	x		
PUT	/Polls/{id}/vote/{optionId}	x		
DELETE	/Polls/{id}	x		
POST	/Polls	x		



Kuva 15. Azuressa olevan Funktion onnistunut kutsu Postmanissa.

Myös AWS Lambda funktiot ja Google Cloud Functions-funktiot testattiin kuvan 15. mukaisesti.

Taulukko 4. Testauksen tulokset AWS Lambda ympäristössä

HTTP-pyyntö	Polku	Toimii odotetusti	Toimii osittain	Ei toimi
GET	/Polls	x		
GET	/Polls/{id}	x		
PUT	/Polls/{id}/vote/{optionId}	x		
DELETE	/Polls/{id}	x		
POST	/Polls	x		

Taulukko 5. Testauksen tulokset Google Cloud Functions ympäristössä

HTTP-pyyntö	Polku	Toimii odotetusti	Toimii osittain	Ei toimi
GET	/Polls	x		
GET	/Polls/{id}	x		
PUT	/Polls/{id}/vote/{optionId}	x		
DELETE	/Polls/{id}	x		
POST	/Polls	x		

Tuloksista voidaan päätellä, että migraatiot jokaiseen serverless-ympäristöön onnistuivat hyvin. Jokaisessa toteutuksessa funktiot toimivat odotetulla tavalla, eikä ongelmia ilmennyt myöskään business-logiikan ulkopuolella. Vaikka infrastruktuuri sovelluksen suoritusympäristössä muuttui, API-kerrosta lukuun ottamatta muihin sovelluksen kerroksiin ei tarvinnut tehdä muutoksia.

Esimerkiksi EF Core ja AutoMapper toimivat suoraan eikä virheitä niiden suhteen ilmennyt, koska kaikki riippuvuudet ja kerrokset, joita funktiot käyttävät, on määritelty Startup.cs-tiedostossa. Näin ollen suurimmat muutokset kohdistuivatkin itse API-kerrokseen, joka korvattiin funktioilla.

Vaikka API olikin toiminnoiltaan suhteellisen yksinkertainen, onnistunut migraatio antaa luottamusta sille, että vastaava migraatio voidaan suorittaa myös API:eille, joissa on enemmän toiminnallisuuksia. Esimerkkisovelluksessa ei

ollut toteutettu autentikointia ja tiedostonkäsittelyä, vaikka nämä ovat yleisiä toiminnallisuuksia REST-API:ssa.

Koska funktioinstansseja luodaan ja poistetaan tarpeen mukaan, näiden toimintojen migraatio ei välttämättä ole yhtä suoraviivaista, ja vaatii todennäköisesti enemmän suunnittelua ja tukeutumista ulkoisiin palveluihin. Näin ollen koodimuutoksia joutuu todennäköisesti tekemään enemmän tiedostojenkäsittelyn ja autentikoinnin osalta.

AWS Lambda, Google Cloud Functions ja Azure Functions kuitenkin tarjoavat kaikki omat palvelunsa tiedostojen lataamiseen ja tallentamiseen, sekä autentikoinnille heidän ympäristöissään. Näitä palveluita ovat esimerkiksi Amazon S3 (Amazon n.d.) tai Azure Storage (Microsoft, 2023g), Azure App Service (Microsoft, 2023f) ja Amazon Cognito (Amazon n.d.). Näiden palveluiden avulla edellä mainitut toiminnot on mahdollista toteuttaa myös serverless-ympäristössä toimivaan REST-API:iin.

7 LOPUKSI

Tässä opinnäytetyössä oli tarkoitus perehtyä serverless-arkkitehtuuriin, sekä siihen kuuluvaan FaaS-palvelumalliin ja sen infrastruktuuriin. Työn päätavoitteena oli siirtää .NET Core:lla toteutettu REST-API kolmen eri palveluntarjoajan serverless-ympäristöihin, sekä käydä läpi mitä muutoksia ohjelmaan piti tehdä koodillisesti, jotta se toimii myös serverless-ympäristössä.

Haastavinta tässä työssä oli selvittää oikeaoppiset konfiguraatiot serverless-ympäristöä varten. Tämä johtui siitä, että esimerkkisovelluksena toiminut API oli alun perin kehitetty eri infrastruktuurille, ja sen keskeiset toiminnot oli jo rakennettu ennen serverless-ympäristöön siirtymistä. Prosessia olisi todennäköisesti helpottanut, jos työssä olisi keskitytty vain yhteen serverless-

ympäristöön, koska jokaisella palveluntarjoajalla on omat vaatimuksensa serverless-sovelluksien konfigurointiin.

Toisaalta prosessia helpotti se, että alkuperäinen API oli toteutettu kerrosarkkitehtuurilla, jolloin esimerkiksi logiikka-, tai tietokantakerrokseen ei tarvinnut koskea. Muutokset painottuivat lähinnä kontrollerien korvaamisen funktiolla, sekä varsinaisiin serverless-konfiguraatiohin,

Migraation tulokset osoittivat, että CRUD-toiminnallisuuksilla toteutettu REST-API voidaan siirtää serverless-ympäristöön suhteellisen pienillä muutoksilla. Tässä korostui myös se asia, että migraatiota tehdessä on tärkeää kiinnittää huomiota serverless-konfiguraatioon, jotta sovellus toimii halutulla tavalla ja oikein. Koska serverless-ympäristöihin siirrettävä API ei sisältänyt laajempia toimintoja, kuten esimerkiksi autentikaatiota, jäi työssä selvittämättä, mitä muutoksia näiden toimintojen siirtäminen serverless-ympäristöön olisi vaatinut.

Yhteenvetona voidaan todeta, että vaikka migraatio voi olla monimutkaisempi API:eille, joissa on enemmän toiminnallisuuksia, se on silti mahdollista hyvällä suunnittelulla toteuttaa, koska palveluntarjoajat tarjoavat työkaluja ja palveluita laajempien toiminnallisuuksien toteuttamista varten kuten edellisessä kappaleessa tuli todettua.

Tätä opinnäytetyötä tehdessä sain hyvän perustietämyksen serverless-arkkitehtuurista, ja sen toiminnasta. Lisäksi opin myös käytännön tasolla miten perinteisessä palvelinympäristössä kehitetty sovellus voidaan siirtää serverless-ympäristöön, kuten ohjelmakoodin tarvittavat refaktoroinnit sekä serveles-ympäristön konfiguraatiot. Nämä opitut käytännön asiat antavat vahvan pohjan tuleville projekteilleni, erityisesti serverless-arkkitehtuurin hyödyntämisen REST-API:ien kehityksessä.

LÄHTEET

Amazon. Using AWS Lambda with Amazon API Gateway - AWS Lambda. Haettu 24.11.2023 osoitteesta https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway.html

Amazon. Using AWS Lambda with Amazon Cognito - AWS Lambda. Haettu 25.11.2023 osoitteesta https://docs.aws.amazon.com/lambda/latest/dg/services-cognito.html

Amazon. What is Amazon S3? - Amazon Simple Storage Service. Haettu 25.11.2023 osoitteesta https://docs.aws.amazon.com/AmazonS3/latest/user-guide/Welcome.html

Astriani, M. (9.11.2020). Serverless: “Pay as You Go”. Haettu 14.2.2023 osoitteesta https://international.binus.ac.id/computer-science/2020/11/09/serverless-pay-as-you-go/

BairesDevBlog. Static vs Dynamic Typing: A Detailed Comparison | Blog - BairesDev. Haettu 1.6.2023 osoitteesta https://www.bairesdev.com/blog/static-vs-dynamic-typing/

Cloudflare. What is vendor lock-in? Vendor lock-in and cloud computing. Haettu 25.2.2023 osoitteesta https://www.cloudflare.com/learning/cloud/what-is-vendor-lock-in/

Cloudflare. Why use serverless computing? Pros and cons of serverless. Haettu 19.2.2023 osoitteesta https://www.cloudflare.com/learning/serverless/why-use-serverless/

Datadog. Serverless Architecture: What It Is & How It Works. Haettu 12.2.2023 osoitteesta https://www.datadoghq.com/knowledge-center/serverless-architecture/

DotNetsExpert. (13.4.2023). History of C# and Its Versions -. Haettu 1.6.2023 osoitteesta https://dotnetsexpert.com/2023/03/13/history-of-c-and-its-versions/

Google. Write Cloud Functions | Cloud Functions Documentation | Google Cloud. Haettu 24.11.2023 osoitteesta https://cloud.google.com/functions/docs/writing

Karim Hafez. Vapauta serverless-arkkitehtuurin voima: hyödyt, haasteet ja parhaat käytännöt. Haettu 2.6.2023 osoitteesta https://www.amabit.fi/post/vapauta-serverless-arkkitehtuurin-voima-hy%C3%B6dyt-haasteet-ja-parhaat-k%C3%A4yt%C3%A4nn%C3%B6t

Lintilä, R. (1.2017). Serverless – mitä se tarkoittaa ja miksi siitä pitäisi kiinnostua? Haettu 12.2.2023 osoitteesta https://www.solita.fi/blogit/serverless-mita-se-tarkoittaa-ja-miksi-siita-pitaisi-kiinnostua/

Microsoft. .NET (and .NET Core) - introduction and overview - .NET | Microsoft Learn. Haettu 22.5.2023 osoitteesta https://learn.microsoft.com/en-us/dotnet/core/introduction

Microsoft. (2023a). Common web application architectures - .NET | Microsoft Learn. Haettu 24.11.2023 osoitteesta https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

Microsoft. (2023b). Guide for running C# Azure Functions in an isolated worker process | Microsoft Learn. Haettu 24.11.2023 osoitteesta https://learn.microsoft.com/en-us/azure/azure-functions/dotnet-isolated-process-guide

Microsoft. (2023c). Supported languages in Azure Functions | Microsoft Learn. Haettu 26.11.2023 osoitteesta https://learn.microsoft.com/en-us/azure/azure-functions/supported-languages?tabs=isolated-process%2Cv4&pivots=programming-language-csharp

Microsoft. (28.2.2023d). Fundamentals of garbage collection - .NET | Microsoft Learn. Haettu 24.11.2023 osoitteesta https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

Microsoft. (25.5.2023e). Overview of Entity Framework Core - EF Core | Microsoft Learn. Haettu 24.11.2023 osoitteesta https://learn.microsoft.com/en-us/ef/core/

Microsoft. (12.10.2023f). Authentication and authorization - Azure App Service | Microsoft Learn. Haettu 25.11.2023 osoitteesta https://learn.microsoft.com/en-us/azure/app-service/overview-authentication-authorization

Microsoft. (12.10.2023g). Introduction to Azure Storage - Cloud storage on Azure | Microsoft Learn. Haettu 25.11.2023 osoitteesta https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction

.NET Handbook. (26.4.2023). .NET Handbook | Serverless / Azure Functions / About Azure Functions. Haettu 25.2.2023 osoitteesta https://infinum.com/handbook/dotnet/serverless/azure-functions/about-azure-functions

Onrego, P. Serverless - mitä tarkoittaa serverless computing? Haettu 12.2.2023 osoitteesta https://onrego.fi/pilviaakkoset/serverless-mita-tarkoittaa-serverless-computing/

Passwater, A. (20.3.2018). When (and why) not to go serverless. Haettu 14.2.2023 osoitteesta https://www.serverless.com/blog/when-why-not-use-serverless

Rehemägi, T. (24.7.2021). Can We Solve Serverless Cold Starts? Haettu 25.2.2023 osoitteesta https://dashbird.io/blog/can-we-solve-serverless-cold-starts/

TIVI. (6.9.2018). Funktioita pilvestä, palvelimettomasti . Haettu 12.2.2023 osoitteesta https://www.tivi.fi/uutiset/funktioita-pilvesta-palvelimetto-masti/bd85e676-c564-39bf-8cc7-ff38418a1d62

Tripathy, S. (25.2.2022). What is Serverless Computing? Benefits & Drawbacks. Haettu 26.2.2023 osoitteesta https://www.enterprisenetworking-planet.com/management/what-is-serverless-computing-benefits-drawbacks/