Mikael Luostarinen

Aaro Leikari

ToF-kameroiden integrointi Azure-pilvipalveluun

eÄlytelli-hankkeen raportti

20. toukokuuta 2021

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Esipuhe

eÄlytelli-hanke (ekologiset, älykkäät ja turvalliset teollisen internetin palvelut) on Euroopan aluekehitysrahaston (EAKR) ja mukana olevien yhteistyöyritysten rahoittama hanke. Hanke keskittyy teollisen internetin tekoälyyn keskittyen kahteen päätutkimusalueeseen: pilvipohjaisten tekoälysovellusten käyttö Big Datan prosessointiin sekä reunalaskentaan. Reunalaskennassa keskitytään datan käsittelemiseen itse IoT-laitteissa tai laiterajapinnassa. Tarkoituksena on kehittää tietoisuutta teollisen internetin keinoälymahdollisuuksista sekä erilaisia tekoälyratkaisuja, jotka kehittävät alueen yritysten liiketoimintaa. Koska tekoäly ja IoT ovat teknologioina vielä alkuvaiheissa, niiden tutkiminen hankkeen avulla tuottaa yrityksille kilpailukykyä. Tässä raportissa keskitytään ToF-tekniikalla toimiviin ihmistenlaskentasensoreihin, niiden asennukseen ja siihen, kuinka ne saadaan lähettämään dataa Microsoft Azuren SQL-tietokantaan.

Termiluettelo

ToF	Time of Flight eli kulkuaikatekniikka.
Terabee	Ranskassa sijaitseva yritys, joka hyödyntää tuotteissaan kulkuaikatekniikkaa.
GUI	Graafinen käyttöliittymä
People Counting L	Terabeen ihmistenlaskentaensori
GDPR	General Data Protection Regulation - EU:n yleinen tietosuoja-asetus
PoE	Power-over-Ethernet - virtalähde, joka vie laitteelle virran lisäksi dataa ethernet-kaapelin välityksellä
Piato	Jyväskylän yliopiston Mattilanniemen Agora-rakennuksessa sijaitseva ravintola.
Tedious	Node-paketti, joka tarjoaa TDS-protokollan toteutuksen Microsoftin SQL-tietokantojen kanssa.
MyJyu	Jyväskylän yliopiston mobiiliaplikaatio, jossa on kalenteri, kampusravintoloiden ruokalistat ja kampusalueen kartta.

Kuviot

Sisällys

1 Johdanto

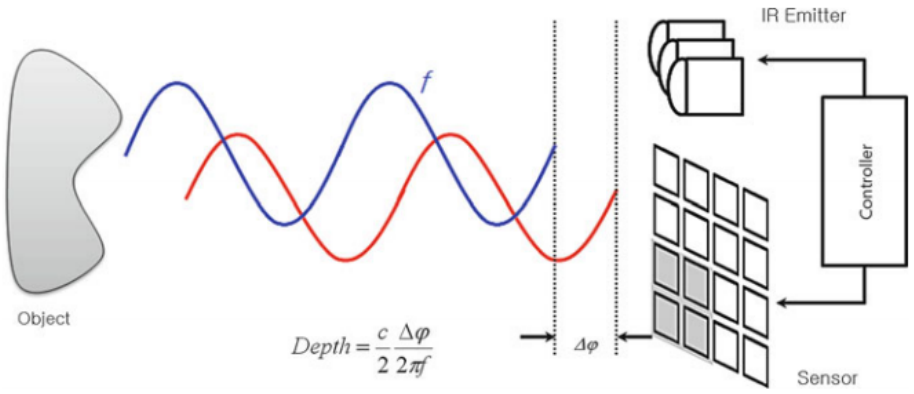
1.1 Raportin sisältö

Tässä raportissa tarkastellaan ToF-kameratekniikkaa ja sen toimintaperiaatteita. Olemme testanneet Terabeen People Counting L - nimistä ihmistenlaskentasensoria, joka toimii ToF-tekniikalla. Raportissa kerrotaan myös Terabeesta ja siitä, mikä People Counting L - sensori on, mitä sillä voidaan tehdä ja kuinka se saadaan otettua käyttöön. Tämän lisäksi raportti sisältää koodin, jolla tämä ihmistenlaskentadata saadaan lähetettyä ja tallennettua Microsoftin Azure-palveluun. Viimeisenä raportissa on yhteenveto.

2 ToF-kamerat

ToF eli time-of-flight-kamerat ovat kulkuaikatekniikalla toimivia kameroita, jotka tuottavat syvyyskuvia. Näissä kuvissa jokainen pikseli kertoo tekoälyn avulla kameralle tiedon siitä, kuinka kaukana se on kuvattavasta kohteesta. Tästä syystä, ToF-kameroita voidaan käyttää arvioimaan 3D-rakennetta suoraan, ilman perinteisiä konenäön algoritmeja. ToF - kamerat toimivat mittaamalla heijastetun infrapunavalon vaiheviiveen.(Hansard ym. 2012.) Tämä käytännössä kertoo kuvan viiveen ajallisesti.

3 Syvyysmittauksen periaatteet



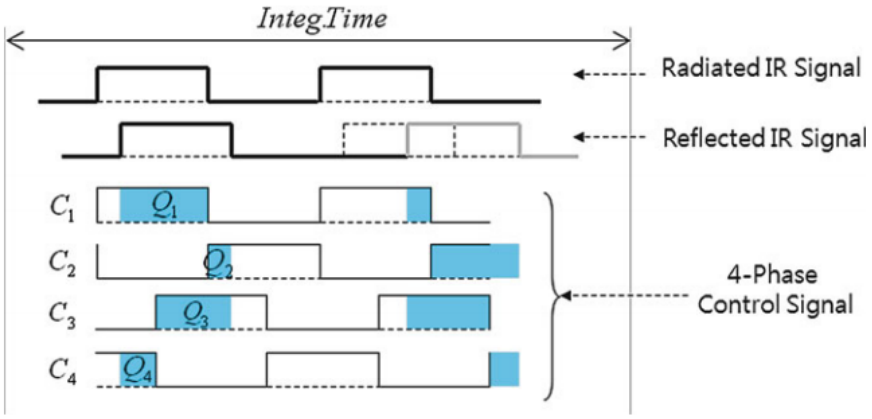
Kuvio 1. Syvyysmittauksen periaate ToF-kameralla.

ToF-kamerat havaitsevat syvyyden infrapunan avulla. Laite lähettää infrapuna-aallon, ja sensori havaitsee takaisin heijastuvan aallon. Aaltojen liikkeeseen kuluneen ajan perusteella voidaan laskea, kuinka kaukana kohde on.(Hansard ym. 2012.) Lähetettyjen ja heijastuneiden infrapuna-signaalien välinen vaiheviive mitataan kohteen etäisyyden laskemiseksi jokaisesta sensorin pikselistä.

$$t_d = arctan\left(\frac{Q3 - Q4}{Q1 - Q2}\right) \tag{3.1}$$

Ylläolevassa laskukaavassa 3.1 näkyy, kuinka vaihe-ero lasketaan neljän vaiheen kontrollisignaaleista. Neljän vaiheen kontrollisignaalilla on 90 asteen vaiheviiveet toisistaan. Ne määrittävät elektronien määrän hyväksytystä infrapunasäteilystä. Neljä tuloksena olevaa sähkövarausarvoa käytetään arvioimaan vaihe-ero *td* siten, että *Q1* - *Q4* edustavat vastaavasti ohjaussignaalien *C1* - *C4* sähkövarauksen määrää. Vastaava etäisyys *d* voidaan sitten laskea alla olevalla kaavalla 3.2, käyttämällä valon nopeutta *c* ja signaalin taajuutta *f*. (Hansard ym. 2012.) Signaalien integraatioaikoja on havainnollistettu kuviossa 2.

$$d = \left(\frac{c}{2f}\frac{t_d}{2\pi}\right) \tag{3.2}$$



Kuvio 2. Signaalien integraatioaika

ToF-kameroilla syvyysmittauksia tehdessä voi tapahtua myös virheitä. Nämä virheet voivat olla joko systemaattisia tai epäsystemaattisia (Foix 2011).

3.1 Systemaattiset virheet

ToF-kameroilla syvyysmittauksia tehdessä voi tapahtua erilaisia systemaattisia virheitä. Virheitä ovat infrapunan demodulaatiovirhe, eli virhe tapahtuu, kun alkuperäinen signaali erotetaan moduloidusta signaalista. Integraatio-aikaan liittyvä virhe tarkoittaa sitä, että samassa paikassa kameralla voidaan saada syvyyden tulokseksi eri luku, mikäli integraatio-aikaa muutetaan. Amplitudin, eli värähdyslaajuuden monitulkintaisuusvirhe johtuu matalista tai liian valottuneista heijastuneista amplitudeista. Tämä johtuu siitä, että syvyyden tarkkuuteen vaikuttaa tulevan valon määrä, josta syvyys voidaan päätellä. Systemaattisiin virheisiin lukeutuvat myös lämpötilaan liittyvät virheet. Kameran lämpötila vaikuttaa kuvan prosessointiin, jonka vuoksi joissakin kameroissa on sisäiset tuulettimet. (Foix 2011.)

3.2 Epäsystemaattiset virheet

ToF-kameroissa on havaittu neljää erilaista epäsystemaattista virhettä. Signaali-melusuhteen vääristymä voi näkyä kohteissa, joita ei ole valaistu tasaisesti. Heikosti valaistut alueet ovat usein enemmän alttiita melulle, kuin hyvin valaistut. Tämän tyyppiset virheet johtuvat usein amplitudista, laitteen parametreista ja kuvattavan alueen syvyyden tasaisuudesta. Usean va-

lon vastaanottovirheet tapahtuvat usean heijastuvan valon häiriöstä, jonka sensorin pikselit vastaanottavat. Nämä heijastukset riippuvat sensorin sivuttaistarkkuudesta ja kuvattavien kohteiden muodosta. Usean valon vastaanottovirheet tapahtuvat usein silloin, kun kohteessa on teräviä tai koveria reunoja. Valonsironta-efekti tapahtuu silloin, kun kameran linssin ja sensorin välillä on paljon valon heijastuksia. Tämä tuottaa syvyyden aliarvioimisen pikseleissä, jotka ovat altistuneet heijastuksille johtuen naapuripikselin heijastusten tuottamasta lisäenergiasta. Valonsironnasta johtuvat virheet ovat merkityksellisiä vain, kun kuvattavalla alueella on kameran lähellä olevia esineitä. Viimeisenä virheenä on liikkeen sumentuminen, joka johtuu kuvattavien objektien tai kameran fyysisestä liikkeestä näytteenottoon käytetyn integraatioajan aikana. (Foix 2011.)

4 Terabee

Terabee perustettiin vuonna 2012 innovatiiviseksi droonipalvelu yhtiöksi. Yhtiön nettisivuilla kerrotaan sen olevan hyvin kansainvälinen ja monimuotoinen organisaatio, joka palkkaa vain kaikkein viisaimmat henkilöt sitä koskettavilta erikoisasiantuntija-aloilta.(“Terabee” 2021.)

4.1 Terabee People Counting L

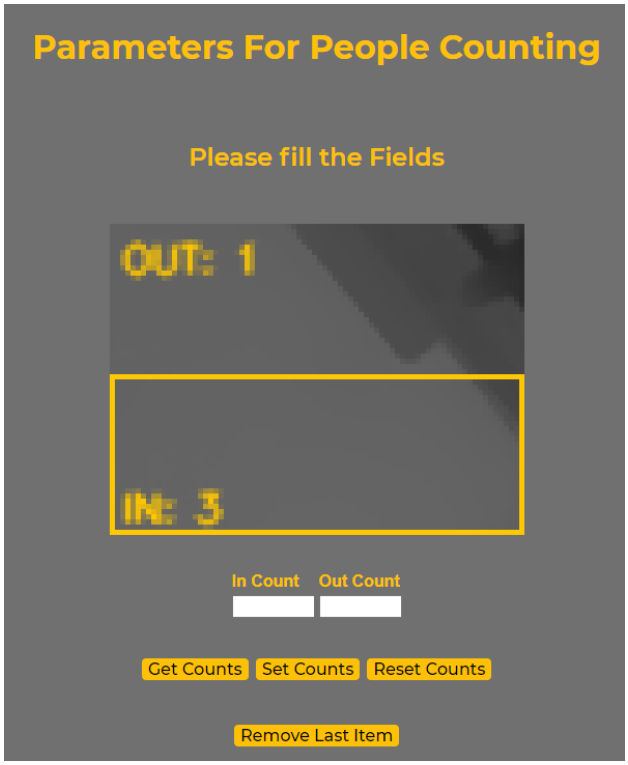
Terabeen People Counting L on ihmistenlaskentasensori, joka toimii ToF-kameratekniikalla. People counting L on kattoon asennettava, hieman palovaroitinta muistuttavan näköinen laite. Nimensä mukaisesti sitä käytetään ihmisten laskemiseen oviaukoissa tai käytävillä. Laskeminen tapahtuu niin, että kun joku liikkuu kuvattavan alueen poikki, laskuri kasvaa. Sensori pystyy havaitsemaan useita ihmisiä yhtäaikaa. Terabee lupaakin vähintään 98 prosentin laskentatarkkuuden. Sensori noudattaa GDPR:ää ja kuvasta ei pystytä tunnistamaan ihmisiä, jolloin anonymiteetti säilyy. Ihmistenlaskentasensorilla voidaan optimoida esimerkiksi tilojen käyttö tai siihen voidaan yhdistää tilan valot, jotka syttyvät kun ihminen menee huoneeseen ja valot sammuvat, kun huone on tyhjentynyt. Tilojen käytön reaaliaikaista ja historiallista dataa voidaan seurata laskurin avulla. (“Terabee People Counting L” 2021) Huomioitavaa on, että laitteen käyttöön vaaditaan PoE-injektori, jolla virta ja data saadaan laitteeseen ja laitteesta ulos ethernet kaapelilla.

4.2 Konfigurointi

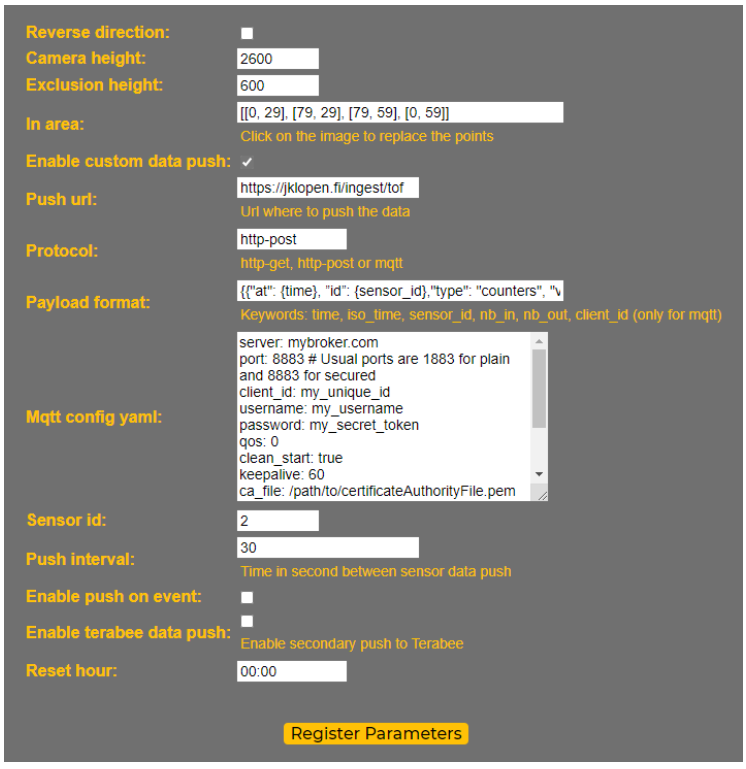
Laitetta pääsee konfiguroimaan sen graafisella käyttöliittymällä internetissä tai älylaitteeseen ladattavalla sovelluksella. Kuviosta 2 näemme, että laitetta konfiguroidessa valitaan kameran kuvasta alue, josta mittaus tapahtuu. Kun joku liikkuu tämän alueen poikki, kasvaa laskuri aina yhdellä. Esimerkiksi, jos laite on oviaukossa ja ovesta liikutaan sisäänpäin, kasvaa laskurin ’IN’ laskuri yhdellä. Samoin, jos liikutaan oviaukosta ulos, kasvaa laskurin ’OUT’ laskuri yhdellä. Halutessa laitteelta voidaan noutaa arvot, asettaa tietyt arvot tai nollata arvot. Seuraavat säädöt ovat nähtävissä kuviossa 3. Kameran ’IN’ ja ’OUT’ mittaussuunnat

Kuvio 3. Terabee People Counting L

voidaan myös vaihtaa toistensa kanssa. Kameran korkeus maasta, sekä mittauksesta pois jätettävä korkeus tulee asettaa millimetreinä tai älylaitesovelluksessa metreinä. Tällä tavalla, jos poissuljettuun korkeuteen on asetettu vaikka arvo 600mm, ei laite laske tätä korkeutta alempana sijaitsevia kohteita mukaan. Sillä pystytään varmistumaan siitä, että laite on laskenut ihmisiä, eikä esimerkiksi kuvausalueen läpi kävelevää koiraa lasketa mukaan. Konfiguroinnissa voidaan myös tietenkin antaa laitteelle 'push url' eli osoite, jonne data lähetetään. Laitteelle voidaan antaa protokollaksi joko http-get,http-post tai mqtt. Mikäli käytetään MQTT:tä, kerättävän datan tietosisällön muotoa voidaan muokata, kuten myös sen YAML-merkintäkielistä sisältöä. Laitteelle voidaan myös asettaa id ja sen push interval:lia eli aikaa datan lähetyksen välissä voidaan muokata. Voidaan myös asettaa push on event, jolloin data lähetetään aina, kun toinen laskureista kasvaa ja data voidaan halutessa lähettää myös Terabee:lle. Viimeisenä asetetaan kellonaika, jolloin laskurit nollaantuvat.



Kuvio 4. Terabeen GUI



Kuvio 5. Terabeen GUI

5 Asennus

Terabee People Counting L ihmistenlaskentasensori asennetaan oviaukon tai käytävän kattoon. Asennuksen yhteydessä täytyy huomioida, ettei laitetta asenneta liian lähelle seinää ja ettei katto ole liian matala tai korkea. Sensorin ja seinän väliin jäävä mitta on riippuvainen katon korkeudesta ja mitat onkin merkitty allaolevaan kuvaan. Asennus Semman Piatoravintolan oviaukoille on hoidettu Aren toimesta.

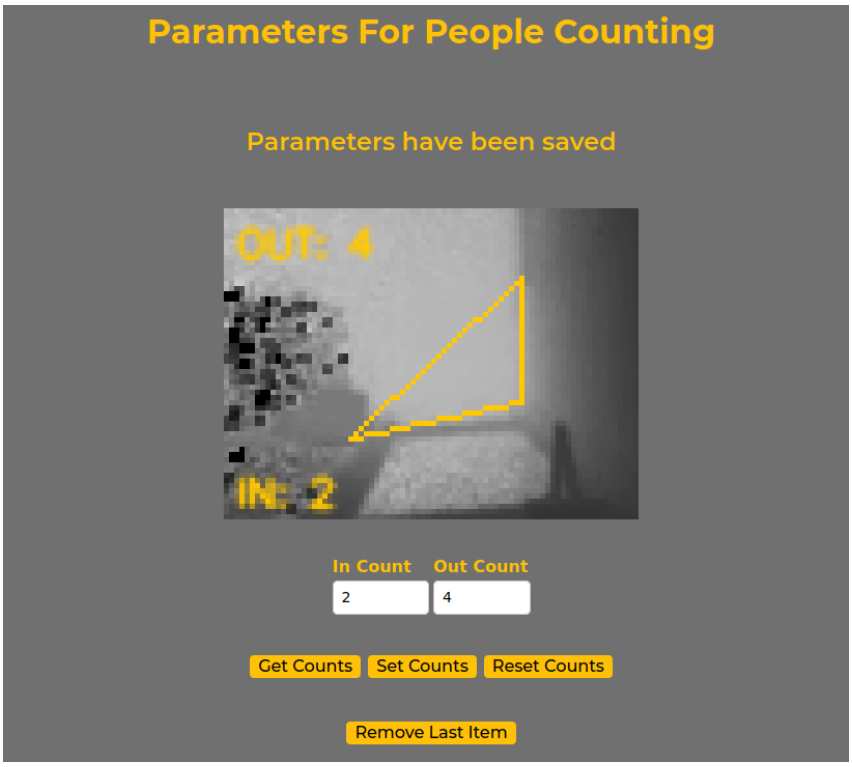
Installation Height (m)	Door or corridor width (m)	Min Clearance from wall (m)
2.40 (Minimum installation Height)	1.50	0.3
2.50	1.55	0.3
2.60	1.60	0.4
2.70	1.70	0.4
2.80	1.80	0.5
2.90	1.90	0.5
3.00	2.10	0.5
3.10	2.20	0.6
3.20 (Maximum installation Height)	2.30	0.6

Kuvio 6. Asennuksen korkeustaulukko

Asennuksen yhteydessä ja sen jälkeen tulee kiinnittää huomiota myös sensorissa oleviin ledeihin. Kuvasta 10 nähdään, mitä mitkäkin värit tarkoittavat. Kuvioissa 7 ja 8 laskentasensorit ovat asennettuina Piaton etu - ja takaovella. Kuviossa 9 nähdään, miltä takaovi näyttää kamerasta.

Kuvio 7. Laskentasensori ravintola Piaton etuovella

Kuvio 8. Laskentasensori ravintola Piaton takaovella



Kuvio 9. Laskentasensorin näkymä Terabeen käyttöliittymässä ravintola Piaton takaovella

LED	Color	Status
Power	Off	Device is not powered
	Green	Device is powered
Status	Off	Device is starting or is off
	Red	Counting software could not start or stopped unexpectedly
	Yellow	Counting software starting
	Green	Counting software started and running
	Blue (brief blink)	Last data pushed successfully
	Blue (constant)	Failing to push data

Kuvio 10. LED-taulukko

6 Terabeen ToF-kameralaskurin ja Microsoft Azuren integraatio

Tässä osiossa käydään läpi yksityiskohtaisesti, kuinka Terabeen People Counting L - ihmistenlaskentasensorista tuleva data saadaan lähetetettyä eteenpäin Microsoftin Azure pilvipalveluun. Alla olevassa kuvassa on esimerkki, millaista JSON-tyyppistä dataa sensori voi lähettää. Kuvassa on myös avainsanojen selitykset englanniksi.

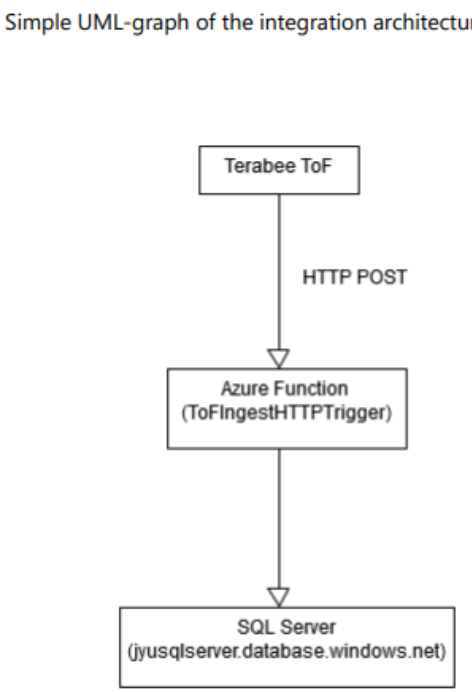
```
{
    "time": {time},
    "id":{sensor_id},
    "value": {
        "in": {nb_in},
        "out": {nb_out}
    }
}
```

ToF keywords explained:

Keyword	Value
time	Unix timestamp
sensor_id	Sensor ID as defined in the web GUI
nb_in	Total current number of entries for the day
nb_out	Total current number of exits for the day

Kuvio 11. Esimerkki JSON datasta, jota ToF-kamera lähettää.

Nyt kun tiedossa on millaista dataa sensori tässä tapauksessa lähettää, keskitytään seuraavaksi siihen, kuinka tuo data saadaan vietyä eteenpäin Azuren tietokantaan. Allaolevassa kuvassa on tehty UML-kaavio tästä arkkitehtuurista. Ensin, Terabeen graafisessa käyttöliittymässä on annettu push-url osoite, jonne data lähetetään. Data lähetetään siis Azureen, jonne on tehty funktio ToFIngestHTTPTrigger (kuva alempana), joka työntää sensorista tulevan datan eteenpäin Jyväskylän yliopiston SQL-tietokantaan.



Kuvio 12. UML-kaavio integraatioarkkitehtuurista.

6.1 Funktioresurssin luominen Azure -pilvipalveluun

Azure -pilvipalveluun on mahdollista luoda funktiota. Näille funktioille voidaan asettaa laukaisijoita, Azuressa tämä nimitys on "trigger"ja tätä nimitystä tullaan käyttämään myös tässä. Kameradataa vastaanottavan funktion trigger on HTTP -tyyppinen, eli funktio aloittaa suorituksen, kun se vastaanottaa sille määritellyn HTTP-pyynnön.

Azureen pystyy helposti luomaan uuden funktion kirjoittamalla portaalissa hakukenttään "function"ja valitsemalla esiin tulevan vaihtoehdon "Function App". Funktion pystyy luomaan "+ Add-nappia painamalla. Painalluksen jälkeen aukeaa uusi näkymä, jossa voi määrittää funktion asetukset. Kun tämä on tehty, painetaa "Review + create", jossa asetukset voi vielä silmäillä läpi ja hyväksyä. Tämän jälkeen funktio on luotu ja valmis käytettäväksi. Nyt funktioon voi lisätä ohjelmakoodia lokaalisti Visual Studiolla tai Visual Studio Codella. Olennaisimpana se, että asennettuna editoriin on Azure laajennus, jolla saa käyttöönsä Azure työkalut. Esimerkiksi Visual Studio Codessa Azure laajennuksen saa haettua helposti "extensions"valikosta kirjoittamalla hakukenttään Azure ja valitsemalla "Azure Tools". Tarkemmat

ohjeet ja kuvat funktion lisäämisestä Azureen löytyvät liitteistä löytyvät liitteistä.

6.2 ToFDataIngest -funktio

Sensorilta saatava data käsitellään Azureen tehdyllä funktiolla. Funktio on kirjoitettu Node.js:llä ja se on HTTPTrigger tyyppinen, eli se aloittaa suorituksen joka kerta, kun se vastaanottaa sille määritellyn HTTP-protokollan mukaisen HTTP-pyynnön, esimerkiksi GET, HEAD tai POST. ToFIngestHTTPTrigger funktiolle määritellyt pyynnöt ovat GET ja POST, mutta tässä tapauksessa GET ei varsinaisesti tee mitään.

Jotta funktiolla voidaan muodostaa tietokantayhteys, käytetään tähän tarkoitukseen Nodelle tehtyä Tedious nimistä pakettia. Tedious on Node paketti, joka tarjoaa implementoinnin Microsoftin SQL-tietokantojen käyttämästä TDS-protokollasta (“Tedious”, n.d.). Tediousista otetaan käyttöön instanssit Connection, Request ja tyypit seuraavasti:

```
// SQL Server connection related dependencies
var Request = require('tedious').Request;
var TYPES = require('tedious').TYPES;
var Connection = require('tedious').Connection;
...
```

Kun funktiolle tehdään POST-pyyntö otetaan pyynnön mukana tulleesta body-objektista data talteen seuraavasti:

```
...
const body = req.body;
const time = body.time;
const id = body.id;
const inCount = body.value.in;
const outCount = body.value.out;
...
```

Tässä ***time*** on lähetyskellonaika, ***id*** on tieto siitä, kummalta sensorilta data on tullut, ***inCount***

on IN-laskurin arvo ja ***outCount*** OUT-laskurin arvo.

Talteen otettu data validoidaan seuraavasti:

```
...
if (time && id && inCount && outCount) {
  ...
} else {
  context.log(new Date().toISOString() + " - Required data
     missing, request has been terminated.");
  context.res = {
    status: 400,
    body: "Required data missing, request has been terminated."
  }
  context.done();
}
...
```

Jos data ei ollut validia, mennään else-lohkon sisälle, tulostetaan virheilmoitus, lähetetään HTTP-protokollan mukainen status 400 ja virheviesti. Lopuksi funktion suoritus lopetetaan.

Jos talteen otetussa datassa oli kaikki kunnossa, mennään if-lauseen sisälle. Ensin luodaan tietokantayhteyttä varten ***config*** niminen objekti, jonka sisälle määritellään tietokanta-autentikaatiota varten vaaditut tiedot:

```
...
var config = {
 server: process.env["SQL_SERVER_DOMAIN"],
 authentication: {
  type: 'default',
  options: {
   userName: process.env["SQL_SERVER_USERNAME"],
   password: process.env["SQL_SERVER_PASSWORD"]
  }
 },
```

```
  options: {
    // If you are on Microsoft Azure, you need encryption:
    encrypt: true,
    database: process.env["SQL_SERVER_DATABASE"]
  }
};
...
```

Tässä ***server*** on tietokantapalvelin, jonne halutaan muodostaa yhteys ja ***authentication*** sisältää autentikointitiedot, joihin sisältyy tyyppi, sekä käyttäjätunnus ja salasana. ***database*** on tietokantapalvelimen tietokanta, jonne halutaan tehdä kyselyt. Azure ympäristössä tarvitaan lisäksi kenttä ***encrypt***.

Seuraavaksi luodaan uusi tietokantayhteys käyttäen aiemmin käyttöön otettua Connection instanssia seuraavasti:

```
var connection = new Connection(config);
connection.on('connect', function(err){
  ...
});
connection.connect();
```

Connection instanssille annetaan parametrina aiemmin luotu config-objekti ja se palauttaa uuden tietokantayhteyden, joka tallennetaan ***connection*** nimiseen muuttujaan. Seuraavaksi määritellään 'connect' niminen tapahtuma, jota kutsutaan ***connectionin*** connect metodilla.

Connect tapahtuman sisällä POST pyynnön mukana tullut data työnnetään tietokantaan. Tietokantaan tallentamisessa käytetään aiemmin käyttöönotettua Request instanssia:

```
var request = new Request("INSERT INTO TerabeeToF
  (ReadingTime, SensorID, InCount, Outcount) VALUES
    (@ReadingTime, @SensorID, @InCount, @OutCOunt);",
      function(err, rowCount){
      if(err){
```

```
            context.log(err);
            res = {
              status: 500,
              body: "Database error, no data has been inserted."
            }
            context.done();
          } else {
            context.log(rowCount + " row(s) returned");
          }
        });

    request.addParameter('ReadingTime', TYPES.Int, time);
    request.addParameter('SensorID', TYPES.TinyInt, id);
    request.addParameter('InCount', TYPES.Int, inCount);
    request.addParameter('OutCount', TYPES.Int, outCount);

    connection.execSql(request);
```

Tässä ***request*** muuttujaan tallennetaan Request instanssin avulla luotu objekti, joka sisältää tietokantakyselyn ja funktion, jolla tehdään tarvittavat toimenpiteet kun kysely on suoritettu. Kyselyn VALUES kohdassa määritellyt arvot täydennetään addParameter-funktion avulla, jolle annetaan parametreina täytettävän kentän nimi, tyyppi, sekä itse arvo. Lopuksi tietokantakysely suoritetaan kutsumalla aiemmin luodun connection-objektin execSql-funktiota ja parametrina annetaan suoritettava tietokantakysely.

Seuraavassa kuvassa on kuvattu, millainen taulukon rakenne on, johon sensorista tullut data lopulta tallennetaan.

The JSON data sent by a ToF camera is saved to the following table:

```
CREATE TABLE dbo.TerabeeToF (
    ID INT IDENTITY(1,1) PRIMARY KEY,
    ReadingTime INT,
    SensorID TINYINT,
    InCount INT,
    OutCount INT
)
```

Kuvio 13. Taulukko, johon kameran lähettämä data tallennetaan.

6.3 ToFCalculator -funktio

Kerätyn datan käyttöä varteen tehtiin HttpTrigger tyyppinen laskentafunktio, jolla voidaan laskea kyseisellä ajan hetkellä ihmisten määrä jossakin tilassa. Tämä tapahtuu yksinkertaisilla vähennys- ja yhteenlaskuilla. Ensin vähennetään molemmilta kameroilta tullut ulos menneiden luku sisään menneiden luvusta. Sitten nämä kaksi lukua summataan yhteen. Tämä yhteen summattu tulos lähetetään eteenpäin.

Kuten tavallista, funktion alussa otetaan käyttöön Noden Tedious kirjasto ja sieltä ***Connection*** ja ***Request*** instanssit. Tämän lisäksi otetaan käyttöön Noden ***axios*** kirjasto, jolla voidaan lähettää HTTP-pyyntöjä.

```
// Require Connection and Request from tedious
const Connection = require("tedious").Connection
const Request = require("tedious").Request;

// Require axios to send post request
const axios = require('axios').default
...
```

Lisäksi luodaan jälleen ***config***-objekti, johon annetaan tarvittavat tiedot tietokantayhteyden muodostamiseksi.

```
...
//Config object used when connecting to database
const config = {
   authentication: {
      options: {
         userName: process.env['SQL_SERVER_USERNAME'],
         password: process.env['SQL_SERVER_PASSWORD']
      },
      type: "default"
   },
   server: process.env['SQL_SERVER'],
   options: {
      database: process.env['SQL_DATABASE'],
      encrypt: true
   }
}
...
```

Tietokantayhteyden luomiseksi on oma funktionsa

```
...
const connectDB = () => {
   return new Promise((resolve, reject) => {
      const connection = new Connection(config)
      connection.connect()
      connection.on('connect', err => {
         if(err) {
            console.log("Failed at connectDB: " + err.message)
            reject(err)
         } else {
            console.log("SUCCESS")
```

```
                resolve(connection)
            }
        })
    })
}
...
```

jossa tuttuun tyyliin käytetään Tediousin ***Connection*** instanssia uuden yhteyden luomiseksi. Instanssille annetaan aiemmin luotu ***config***-objekti. Funktiosta palautetaan uusi tietokantayhteys.

Koko funktio lähtee liikkellee ***try-catch*** -lohkosta

```
...
// Start point of the Azure function
try {
    // call the function to fetch data from database
    // and pass a callback function as parameter
    getCameraData((err, data) => {
        if(err) {
            console.log(err.message)
        }
        else {
            doCalc(data)
        }
    })

} catch (error) {
    console.log(error.message)
}
```

Kun funktiolle tulee ***HTTP***-pyyntö, suoritetaan ensin ***try-catch*** -lohko. Samalla tällä käytännössä hoidetaan virheidenkäsittely. Jos jokin menee vikaan funktion suorituksessa, tul-

laan ***catch*** -lohkoon ja tulostetaan virheilmoitus. ***Try***-lohkossa mennään suorittamaan ***getCameraData*** -funktio. Funktiolle annetaan parametrina ***callback***-funktio, jota kutsutaan kun ***getCameraData*** -funktio on lopettanut suorituksen.

GetCameraData -funktiossa käydään ensin hakemassa uusi tietokantayhteys ***ConnectDB*** -funktiolta, jonka jälkeen suoritetaan SQL-kysely.

```
...
const connection = await connectDB()
await connection.execSql(request)
...
```

Itse kyselylausekkeella haetaan molempien kameroiden sensoreilta tullut data uusimman tiedon mukaan. Kysely suoritetaan seuraavasti:

```
    ...
    // make the request
    const request = await new Request(query, async (err,
       rowCount) => {
       // give the fetched data or error to the callback function
       if(err) {
          console.log("Failed at getCameraData: ")
          callback(err, null)
       }
       else {
          console.log("FETCHED " + rowCount + "ROW(S)")
          callback(null, data)
          connection.close()
       }
    })

    // create object from each rows column name and value
    // and store the object to data variable
    await request.on('row', columns => {
```

```
    columns.forEach(row => {
        data.push({name: row.metadata.colName, value:
            row.value})
    })
})
...
```

Yllä olevat rivit suoritetaan jossain määrin asynkronisesti, joten tulee olla tarkkana, että jokaisessa kriittisessä kohdassa odotetaan suorituksen lopettamista. Funktiossa ***getCameraData*** luodaan ***query*** niminen muuttuja, jonne asetetaan tietokantakysely, jolla haetaan SQL-tietokannasta molemmilta kameroilta saadut uusimmat tiedot sisään menneiden ja ulos tulleiden ihmisten lukumäärästä. Itse kysely suoritetaan Tediousin ***Request*** -metodilla, josta palautetaan kysely ***request*** muuttujaan.

Tämän jälkeen ***request*** muuttuja saa käyttöönsä paljon metodeita, joista ***row*** -metodia käytetään, kun otetaan talteen tietokannasta halutut rivit. Tietokantarivit tallennetaan objektina ***data*** nimiseen taulukkoon, jonne tallennetaan riviltä ***name*** -kenttään sensorin numero, sekä ***value*** -kenttään itse arvo.

```
...
// create object from each rows column name and value
// and store the object to data variable
await request.on('row', columns => {
    columns.forEach(row => {
        data.push({name: row.metadata.colName, value:
            row.value})
    })
})
...
```

Kun tämä on tehty aletaan suorittaa ***callback*** -funktiota, jolle annetaan parametrina sensoriarvoilla täydennetty ***data*** -taulukko.

```
...
// give the fetched data or error to the callback function
   if(err) {
      console.log("Failed at getCameraData: ")
      callback(err, null)
   }
   else {
      console.log("FETCHED " + rowCount + "ROW(S)")
      callback(null, data)
      connection.close()
   }
})
...
```

Jos funktion suorituksessa sattui virhe, tehdään tulostus ja palautetaan ***callback*** -funktion mukana virheilmoitus. Muussa tapauksessa tulostetaan montako riviä haettiin, palautetaan ***callback*** -funktion mukana data ja suljetaan tietokantayhteys.

Kun ollaan päästy ***getCameraData*** -funktiosta, palataan takaisin Azure funktion "pääsilmukkaan"***callback*** -funktion suoritukseen.

```
...
// call the function to fetch data from database
// and pass a callback function as parameter
getCameraData((err, data) => {
   if(err) {
      console.log(err.message)
   }
   else {
      doCalc(data)
   }
})
...
```

Jos ***getCameraData*** -funktiolta palautui virhe, tulostetaan virheilmoitus ja lopetetaan suoritus. Muussa tapauksessa kutsutaan ***doCalc*** -funktiota, jossa suoritetaan itse laskutoimitus ja jolle annetaan parametrina ***getCameraData*** -funktion palauttama data.

```
...
// Do the calculation here
const doCalc = (data) => {
    const sensor1 = data[0].value - data[1].value
    const sensor2 = data[2].value - data[3].value
    const sum = sensor1 + sensor2
    console.log(sum)

    sendData(sum)
}
...
```

doCalc -funktio on melko yksinkertainen. Molempien sensoreiden ulosmenneiden ihmisten määrä vähennetään sisäänmenneiden ihmisten määrästä ja näistä molemmista vähennyslaskuista saadut tulokset summataan yhteen, jolloin saadaan tieto siitä, montako ihmistä on tietyllä ajan hetkellä sisällä jossakin tilassa.

```
...
// Send data to myjyu app
const sendData = (sum) => {
    console.log("Send post request here.")
    axios.post('/some_endpoint', {
        sum: sum
    }).then((response) => {
        console.log(response)
    }).catch((error) => {
        console.log(error.message)
    })
}
...
```

sendData -funktiossa laskutoimituksen tulos lähetetään haluttuun paikkaan, tässä tapauksessa **MyJyu** -sovellukseen. Lähettämiseen käytetään tunnetusti **POST** -metodia ja lähetyksen tekemiseen käytetään aiemmin käyttöön otettua axiosia. Kun data on lähetetty, käsitellään mahdollinen paluuarvo, joka lähetyksen jälkeen saadaan. Tässä tapauksessa tosin vastaus vain tulostetaan. ***ToFCalculator*** -funktio löytyy kokonaisuudessaan liitteistä.

7 Yhteenveto

ToF-kameratekniikalla voidaan seurata helposti ihmisten liikkumista halutussa ympäristössä identifioimatta ihmisiä, jolloin yksityisyydensuojan kanssa ei tule ongelmia. Tämän tyyppinen projekti oli aivan uudenlainen meille Jyväskylän yliopistolla, joten tässä raportissa esiteltyä materiaalia ja koodia voidaan tulevaisuudessa hyödyntää saman tyyppisissä projekteissa. Tämän kyseisen projektin dataa tullaan käyttämään Jyväskylän Yliopiston Agora-rakennuksessa sijaitsevan Piato-ravintolan asiakkaiden laskennassa. Tieto siitä, kuinka monta ihmistä ravintolassa on kyseisellä hetkellä sisällä lähetetään MyJYU-appiin, josta tilannetta voi seurata kuka vain.

Lähteet

Foix, Torras, Alenya. 2011. “Lock-in Time-of-Flight (ToF) Cameras: A Survey”, https://core.ac.uk/download/pdf/41765475.pdf.

Hansard, Miles, Seungkyu Lee, Ouk Choi ja Radu Patrice Horaud. 2012. *Time-of-flight cameras: principles, methods and applications.* Springer Science & Business Media.

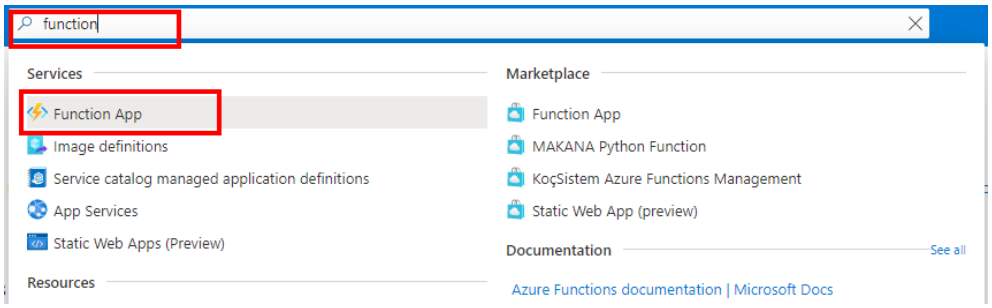
“Tedious”. n.d. https://tediousjs.github.io/tedious/index.html.

“Terabee”. 2021. https://www.terabee.com/about/.

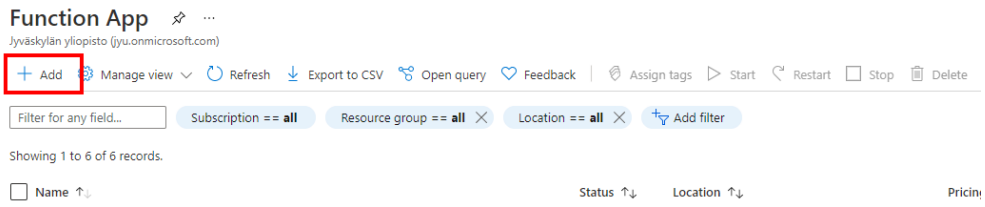
“Terabee People Counting L”. 2021. https://www.terabee.com/shop/people-counting/terabee-people-counting-l/.

Liitteet

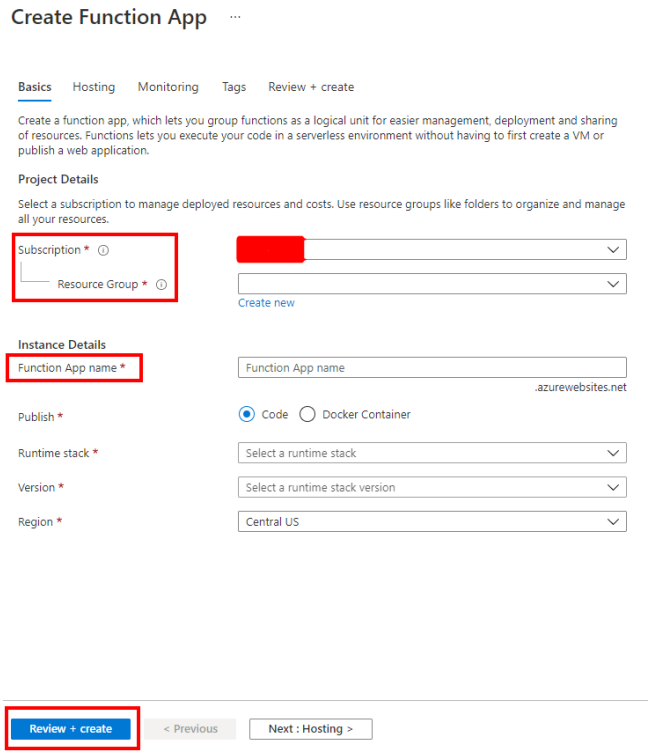
A Funktion lisääminen Azure -pilvipalveluun



Kuvio 14. Syötetään hakukenttään "function"hakusana ja valitaan "Function App".



Kuvio 15. Valitaan avautuvasta näkymästä, sivun ylälaidasta "+Add".



Kuvio 16. Määritetään funktion asetukset. Lopuksi lisätään funktio painamalla "Review + create".

B ToFIngetHTTPTrigger -funktio

```
// SQL Server connection related dependencies
var Request = require('tedious').Request;
var TYPES = require('tedious').TYPES;
var Connection = require('tedious').Connection;

module.exports = function (context, req) {
  context.log('JavaScript HTTP trigger function processed a
     request.');

  const body = req.body;
  const time = body.time;
  const id = body.id;
  const inCount = body.value.in;
```

```
const outCount = body.value.out;

const responseMessage = "Received ToF data push from sensor ID:
   " + id + " at " + time + " with following values: in " +
   inCount + ", out " + outCount + ".";
context.log(responseMessage);

if (time && id && inCount && outCount) {
  try {
    var config = {
      server: process.env["SQL_SERVER_DOMAIN"],
      authentication: {
        type: 'default',
        options: {
          userName: process.env["SQL_SERVER_USERNAME"],
          password: process.env["SQL_SERVER_PASSWORD"]
        }
      },
      options: {
        encrypt: true,
        database: process.env["SQL_SERVER_DATABASE"]
      }
    };
    var connection = new Connection(config);
    connection.on('connect', function(err) {
      context.log(new Date().toISOString() + " - Attempting to
         connect to SQL Server");
      if (err) {
        context.log.error(err);
        context.res = {
          status: 500,
          body: "Error connecting to Azure SQL Server."
        };
```

```
    context.done();
} else {
    context.log("Successfully established database
        connection. Inserting data - " + new
        Date().toISOString());
    var request = new Request("INSERT INTO TerabeeToF
        (ReadingTime, SensorID, InCount, OutCount) VALUES
        (@ReadingTime, @SensorID, @InCount, @OutCount);",
        function(err, rowCount) {
      if(err) {
        context.log.error(err);
        res = {
          status: 500,
          body: "Database error, no data has been
              inserted."
        }
        context.done
      } else {
        context.log(rowCount + " row(s) returned");
      }
    });
    request.addParameter('ReadingTime', TYPES.Int, time);
    request.addParameter('SensorID', TYPES.TinyInt, id);
    request.addParameter('InCount', TYPES.Int, inCount);
    request.addParameter('OutCount', TYPES.Int, outCount);

    connection.execSql(request);
    context.log("Successfully inserted sensor data at: " +
        new Date().toISOString());
    context.res = {
      status: 200,
      body: "Successfully inserted sensor data."
    };
```

```
            context.done();
          }
        });
        connection.connect();
      } catch (error) {
        context.log(new Date().toISOString() + " - Database
            error.");
        res = {
          status: 500,
          body: "Database error, no data has been inserted."
        }
        context.done();
      }

    } else {
      context.log(new Date().toISOString() + " - Database error.");
      res = {
        status: 400,
        body: "Required data missing, request has been terminated."
      }
      context.done();
    }

}
```

C ToFCalculator -funktio

```
/**
 * Azure function with timer trigger to calculate amount of
    people
 */
```

```
// Require Connection and Request from tedious
const Connection = require("tedious").Connection
const Request = require("tedious").Request;

// Require axios to send post request
const axios = require('axios').default

module.exports = async function (context, myTimer) {
    var timeStamp = new Date().toISOString();

    if (myTimer.isPastDue)
    {
        context.log('JavaScript is running late!');
    }
    context.log('JavaScript timer trigger function ran!',
        timeStamp);

    //Config object used when connecting to database
    const config = {
        authentication: {
            options: {
                userName: process.env['SQL_SERVER_USERNAME'],
                password: process.env['SQL_SERVER_PASSWORD']
            },
            type: "default"
        },
        server: process.env['SQL_SERVER'],
        options: {
            database: process.env['SQL_DATABASE'],
            encrypt: true
        }
    }
```

```
// Function to fetch a new connection to database
const connectDB = () => {
   return new Promise((resolve, reject) => {
      const connection = new Connection(config)
      connection.connect()
      connection.on('connect', err => {
         if(err) {
            console.log("Failed at connectDB: " + err.message)
            reject(err)
         } else {
            console.log("SUCCESS")
            resolve(connection)
         }
      })
   })
}

// Function to get the rows from database
const getCameraData = async (callback) => {

   // use this query
   const query = "SELECT Sensor1.InCount, Sensor1.OutCount FROM
      (SELECT TOP 1 InCount, OutCount FROM TerabeeToF WHERE
      SensorID = 1 ORDER BY ReadingTime DESC) Sensor1 UNION ALL
      SELECT Sensor2.InCount, Sensor2.OutCount FROM (SELECT TOP
      1 InCount, OutCount FROM TerabeeToF WHERE SensorID = 2
      ORDER BY ReadingTime DESC) Sensor2"

   // variable where the fetched data is stored
   let data = []

   // make the request
```

```
    const request = await new Request(query, async (err,
        rowCount) => {
        // give the fetched data or error to the callback function
        if(err) {
            console.log("Failed at getCameraData: ")
            callback(err, null)
            connection.close()
        }
        else {
            console.log("FETCHED " + rowCount + "ROW(S)")
            callback(null, data)
            connection.close()
        }
    })

    // create object from each rows column name and value
    // and store the object to data variable
    await request.on('row', columns => {
        columns.forEach(row => {
            data.push({name: row.metadata.colName, value:
                row.value})
        })
    })

    //get new connection instance and execute the request
    const connection = await connectDB()
    await connection.execSql(request)

}

// Do the calculation here
const doCalc = (data) => {
    const sensor1 = data[0].value - data[1].value
```

```
    const sensor2 = data[2].value - data[3].value
    const sum = sensor1 + sensor2
    console.log(sum)

    sendData(sum)
}

// Send data to myjyu app
const sendData = (sum) => {
    console.log("Send post request here.")
    axios.post('/some_endpoint', {
        sum: sum
    }).then((response) => {
        console.log(response)
    }).catch((error) => {
        console.log(error.message)
    })
}

// Start point of the Azure function
try {
    // call the function to fetch data from database
    // and pass a callback function as parameter
    getCameraData((err, data) => {
        if(err) {
            console.log(err.message)
        }
        else {
            doCalc(data)
        }
    })

} catch (error) {
```

```
        console.log(error.message)
    }
};
```