jamk

Automatisoitu käyttäjänhallinta (IAM)

Ville Valkila

Opinnäytetyö, AMK
Elokuu 2023
Tietojenkäsittely ja tietoliikenne
Tieto- ja viestintätekniikan tutkinto-ohjelma (AMK)

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Valkila, Ville

Automatisoitu käyttäjänhallinta (IAM)

Jyväskylä: Jyväskylän ammattikorkeakoulu. Elokuu 2023, 34 sivua.

Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Tavoitteena oli tutustua IAM-järjestelmien keskeisiin ominaisuuksiin ja kehittää kevyt ja tehokas automatisoitu ratkaisu käyttäjän- ja pääsynhallinnan integroimiseen ERP-järjestelmän ja Active Directoryn sekä Azure AAD:n välillä. Toimeksiantajana oli Qvantel Finland Oy, jolla oli käytössään Midpointin IAM-järjestelmä. Alkuperäinen järjestelmä oli kuitenkin raskas ja kallis, joten toiveena oli tutkia ja kehittää mittatilaustyönä toimivaa järjestelmää, joka integroitaisiin jo valmiiksi käytössä oleviin ympäristöihin.

Toteutus tehtiin kahdella eri Python-prosessilla joista toinen synkronoi ERP-järjestelmän päivittämästä tietokannasta muutoksia ja tallentaa ne tehtäviksi Azure Service Bus:n tehtäväjonoon. Toinen prosessi käy läpi näitä tehtäviä ja tekee Active Directoryyn tarvittavat muutokset ja merkkaa tehtävät valmiiksi. Tavoitteet täyttyivät ja tuloksena saatiin toimiva ja yksinkertainen ratkaisu, josta voidaan jatkokehittelyllä saada tuotantovalmis järjestelmä, joka helpottaa ja keskittää identiteetinhallintaa.

Identiteetin- ja pääsynhallinnan automatisointi ja keskittäminen niille tarkoitettuihin järjestelmiin helpottaa huomattavasti sekä loppukäyttäjien, sekä IT-henkilöstön työarkea, sillä loppukäyttäjien vastuulle jää muistaa käyttäjätunnuksia ja salasanoja vain muutamiin, tai mahdollisesti vain yhteen, järjestelmiin. Myös järjestelmistä vastuussa oleville henkilöille työtaakka helpottuu, kun tunnuksia ei tarvitse tehdä jokaiseen eri palveluun, vaan ne voidaan hoitaa integraatioilla keskitettyyn identiteetinhallintapalveluun. Keskitetystä IAM-järjestelmästä on hyötyä myös tietoturvan kannalta, sillä on helpompaa pitää turvassa yksi tili, joka vaatii vahvaa tunnistautumista, kuin huolehtia monesta eri tilistä, joista voi olla vaikea pysyä perässä. Myös pääsynhallinnan auditointi ja tietoturvariskit pienenevät, kun esimerkiksi poistuvia käyttäjiä hallinnoidaan automaation avulla.

Avainsanat (asiasanat)

IAM, Python, Docker, Kubernetes, Active Directory, Azure, järjestelmäkehitys

Muut tiedot (salassa pidettävät liitteet)

jamk

Description

Valkila, Ville

Automating identity management (IAM)

Jyväskylä: JAMK University of Applied Sciences, August 2023, 34 pages.

Degree Programme in Information and Communication Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

Objective was to get to know the essential functions of an IAM system and to develop a light and optimized automated solution to integrate ERP-system with Active Directory and Azure AAD with custom made identity and access management system. Assignment was created by Qvantel Finland Oy which had Midpoint IAM system in use, but it was too heavy and expensive for their use case. They wanted to investigate a possibility of developing custom-made system to integrate their different environments.

Implementation was created with two different Python processes. The first one syncs data from ERP-systems database and finds changes which need to be updated to Active Directory environment, and then creates tasks of these updates to an Azure Service Bus queue. Another process goes through the tasks in the queue, creates the needed updates in Active Directory and marks the tasks done after. These objectives were met, and the result was a working system which was simple and effective. With some further development it would be possible to enhance this to a production ready system to help with centralizing and automating identity management.

Automating identity and access management will ease the working life of employees and IT staff, as the end users will have to remember only few, or possibly just one, different credentials. Work for IT staff will be easier as all the different services and systems don’t need their own user accounts to be created, but they can be handled with one centralized identity and integrations. Centralized IAM system will also help with cyber security as it is easier to keep one identity safe with strong identification, as it would be to keep up with multiple accounts and credentials. Also auditing and keeping information secure with access management gets easier, as for example access of leaving employees can be terminated automatically by the system.

Keywords/tags (subjects)

IAM, Python, Docker, Kubernetes, Active Directory, Azure, system development

Miscellaneous (Confidential information)

Sisältö

Kuviot

Lyhenteet

AAD	Azure Active Directory
ACR	Azure Container Registry
AD	Active Directory
AKS	Azure Kubernetes Service
API	Application Programming Interface
AWS	Amazon Web Services
ERP	Enterprise Resource Planning (toiminnanohjausjärjestelmä)
HR	Human Resources
IAM	Identity and Access Management
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
LDAPS	LDAP over SSL
RDP	Remote Desktop Protocol
SSL	Secure Sockets Layer
TLS	Transport Layer Security

1 Johdanto

1.1 Opinnäytetyön tausta ja tavoitteet

Tietoturvavaatimukset kasvavat koko ajan digitalisaation leviämisen myötä. Yksi suurimmista tietoturvaheikkouksista on käyttäjien todentaminen ja valtuuttaminen. Jos käyttäjä voi esiintyä toisena henkilönä ja sitä kautta saa käyttöönsä tuon toisen käyttäjän digitaaliset oikeudet, on väärinkäytön mahdollisuus suuri. Tästä voi koitua suurta haittaa tuolle käyttäjälle itselleen tai vastaavasti organisaatiolle, joka on antanut käyttäjälle toimintavaltuuksia omiin järjestelmiinsä. Yksi mahdollisuus vähentää tähän liittyviä riskejä on keskittää ja automatisoida käyttöoikeuksia ja todentamista mahdollisimman paljon yhteen järjestelmään, josta oikeuksia hajaannutetaan eteenpäin eri järjestelmiin. Näin tuota yhtä järjestelmää voidaan koventaa, hallinnoida ja valvoa tehostetummin ja voidaan varmistua siitä, että oikeilla ihmisillä on oikeat pääsyoikeudet järjestelmiin. Tallaista keskitettyä järjestelmää kutsutaan identiteetin ja pääsynhallinnaksi, eli IAM:ksi, johon tämä opinnäytetyö keskittyy.

Opinnäytetyön toimeksiantajalla Qvantel Finland Oy:lla on nykyisessä käytössä Midpointin IAM-järjestelmä. Nykyinen järjestelmä on kuitenkin raskas ja hidas, sekä lisenssimaksuiltaan kallis. Nykyisiin tarpeisiin riittäisi yksinkertaisempi ja kevyempi ratkaisu, joten toimeksiannoksi annettiin kehittää yrityksen omiin tarpeisiin mukautettu sisäinen IAM-ratkaisu, joka hallinnoisi automaattisesti käyttäjä- ja pääsynhallinnan HR-järjestelmässä tapahtuvien muutosten perusteella. Kehitettävän järjestelmän päätavoitteena oli hallinnoida työntekijöiden käyttäjätilejä ja pääsyoikeuksia sen mukaan mitä tietoja heistä on HR-järjestelmään syötetty. Uusille työntekijöille oli tavoitteena luoda uudet käyttäjätilit ja esimerkiksi työntekijöiden yhteystietojen muuttuessa järjestelmän tavoitteena oli päivittää nuo tiedot myös digitaalisille käyttäjätileille. Toisaalta tavoitteena oli myös varmistaa, että työsuhteen päättyessä järjestelmästä poistetaan työntekijän pääsyoikeudet ja käyttäjätilit kytketään pois päältä. Lopuksi haluttiin, että jos käyttäjän tili on ollut kuukauden kytkettynä pois päältä, poistetaan tili lopullisesti.

1.2 Qvantel Finland Oy

Asiakastieto sivuston (Qvantel Finland Oy n.d.) mukaan Qvantel Finland Oy on vuonna 1990 perustettu ohjelmistoyritys, joka tuottaa pääasiassa pilvipohjaisia BSS-palveluja (Business Support Sys-

tems) teleyrityksille. Yrityksen toimitusjohtaja on Matti Roto. Vuonna 2021 yrityksellä oli henkilöstöä noin 240 ja liikevaihto oli 53,0 miljoonaa euroa. Qvantelin (About Qvantel 2023) omien tietojen mukaan Qvantel on kansainvälinen yritys, jolla on toimipaikkoja Suomen lisäksi mm. Espanjassa ja Intiassa. Asiakkaita löytyy 23 eri maasta ja Qvantel käsittelee yli 230 miljoonan teleoperaattoriasiakkaan tietoja. Suomessa Qvantel Finland Oy:llä on toimipaikkoja Jyväskylässä, Helsingissä ja Tampereella. Yrityksellä on Nixu Certification Oy:n myöntämä ISO 27001 sertifikaatti.

2 Tutkimusasetelma

2.1 Tutkimuskysymykset

Tämän tutkimuksen tarkoituksena on kartoittaa onko mahdollista luoda itse kevyempi versio nykyisestä IAM-järjestelmästä ja jos se on mahdollista, kehittää toimeksiantajalle sopiva ratkaisu. Tutkimuksessa käytetään kehittämistutkimusta. Tutkimuksessa pyritään vastaamaan seuraaviin tutkimuskysymyksiin ja niiden pohjalta luomaan sovellus, joka ottaa huomioon kysymysten käsittelyssä ilmenevät tarvittavat ominaisuudet:

1. Miten automatisoida ERP-järjestelmän ja identiteetinhallinnan integraatio
2. Miten varmistua siitä, että IAM-järjestelmän tiedot ovat ajantasaiset
3. Kuinka tehdä IAM-järjestelmästä vikasietoinen, kevyt ja tehokas

Tutkimuksessa pyritään järjestelmäkehityksen kannalta tutustumaan siihen mitä IAM-järjestelmältä vaaditaan ja sen pohjalta tutkia, onko mahdollista luoda itse järjestelmä, joka tehtäisiin mittatilaustyönä toimeksiantajan tarpeisiin niin, että luodaan automatisoidut integraatiot suoraan tarvittavien tietokantojen ja ympäristöjen välille. Koska käsiteltävä tieto on arkaluontoista ja järjestelmän toiminta vaikuttaa suoraan koko henkilöstön mahdollisuuksiin kirjautua eri järjestelmiin ja työskennellä tehokkaasti, on tutkimuksessa kiinnitettävä erityistä huomiota siihen, että kehitettävä järjestelmä on varmasti ajantasainen henkilöstötietojen osalta, ja että järjestelmä on mahdollisimman vikasietoinen. Toisaalta myös tutkimuksessa on kiinnitettävä huomiota siihen, että tuotettu ratkaisu olisi nopea ja kevyt, sillä toimeksiantajan nykyisessä käytössä olevan IAM-järjestelmän raskaus ja epäluotettavuus ovat suurimmat syyt tämän tutkimuksen toimeksiannolle.

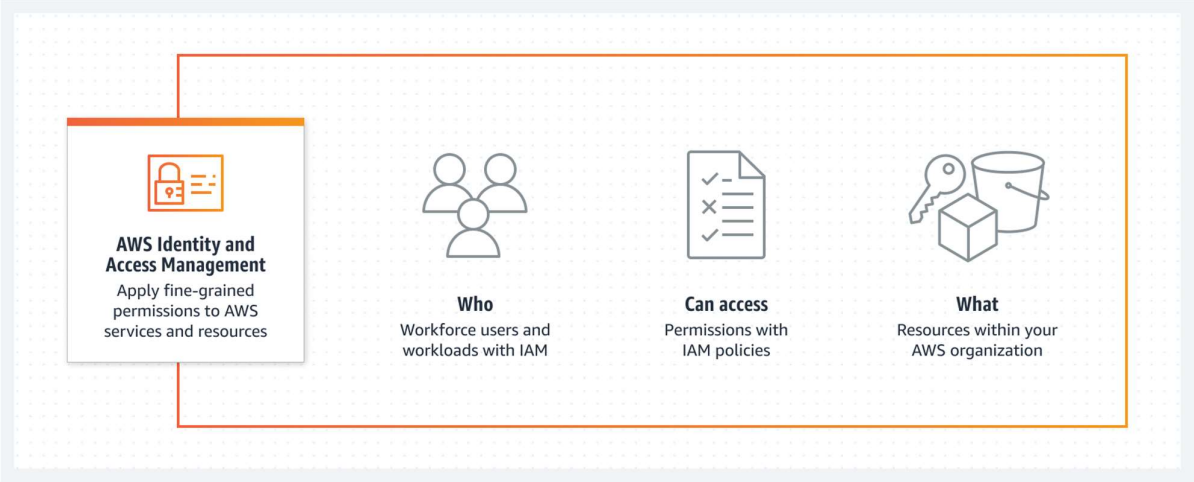
2.2 Kehittämistutkimus

Kehittämistutkimuksessa yhdistyy teoreettiset ja käytännön vaiheet, joita iteroimalla voidaan päästä haluttuun ratkaisuun. Kehitys keskittyy valmiiksi olemassa olevaan teoriaan, mutta tuottaa myös uutta teoriaa. Yksi merkittävä kehittävän tutkimuksen ominaisuus on se, että se johtaa käytettävään käytännön tulokseen, samalla tuottaen tutkimusta edistävää tietoa. Tämä lähestymistapa sopii toimeksiantoon, sillä tavoitteena on pilkkoa olemassa olevassa ympäristössä oleva ongelma pienempiin osiin ja valmiiksi olemassa olevan tiedon ja ympäristön vaatimusten mukaan kehittää sovellusta, joka tarjoaisi sekä ratkaisun kyseisiin ongelmiin. Samalla tuotetaan hyödyllistä tutkimustietoa, mistä muutkin osapuolet voivat hyötyä. (Pernaa 2013.)

3 Identiteetin ja pääsynhallinta (IAM)

3.1 Yleistä

Digitalisoitumisen myötä tietoturvavaatimukset kasvavat nopealla tahdilla. Yksi tietoturvaa helpottavista ja vahvistavista järjestelmistä on IAM, joka tulee englannin kielen sanoista Identity and Access Management, on tietoturvallisuuteen liittyvää käyttäjien ja todennuksen hallintaa, eli käyttäjien sähköisen identiteetin hallintaa. Niemen (n.d.) mukaan IAM vastaa pohjimmiltaan kysymyksiin: kenellä (on oikeus), mihin, miksi ja miten. Amazon Web Services (AWS Identity and Access Management (IAM) n.d.) toisaalta kiteyttää oman IAM järjestelmänsä toiminnallisuuden kysymykseen ”Kuka pääsee mihinkin?” (ks. kuvio 1). IAM:n avulla sovitetaan eri järjestelmät ja teknologiat yhteen niin, että käyttäjillä tai järjestelmillä, jotka tarvitsevat oikeuksia resursseihin, on olemassa yksi sähköinen identiteetti ja tuolla identiteetillä tarvittavat pääsyoikeudet. IAM voidaan jakaa kahteen eri toiminnallisuuteen, identiteetinhallintaan, sekä pääsynhallintaan. Pelkkä identiteetinhallinta on terminä jo noin 20 vuotta vanha ja sillä tarkoitetaan usein käytännössä sitä, että ihmisillä tai resursseilla on oma sähköinen identiteetti, jonka avulla voidaan todeta käyttäjän olevan se, jonka väittääkin olevansa. Tallaiset erilaiset identiteetit voidaan tallentaa ja hallinnoida keskitetysti. Myöhemmin kuitenkin haluttiin pelkkiin identiteetteihin liittää myös pääsyoikeudet, jotta eri järjestelmissä olevat tunnukset ja oikeudet voitaisiin keskittää yhdelle oikeassa elämässä olevalle ihmiselle tai laitteelle, ja tästä liitoksesta syntyi termi IAM. (Niemi n.d.)



Kuvio 1. AWS:n määritelmä IAM:sta (AWS Identity and Access Management (IAM) n.d.).

Identiteetin ja pääsynhallinnan toiminnallisuudet voidaan jakaa neljään eri perustoimintoon. Ensimmäisenä on autentikointi, jonka avulla tunnistetaan, että käyttäjä tai laite on todellakin se kuka tai mikä väittääkin olevansa. Yleisin autentikointitapa on käyttää käyttäjätunnusta ja salasanaa, mutta nykyään autentikoinnin tietoturvaa voidaan parantaa käyttämällä lisäksi esimerkiksi monivaiheista tunnistautumista tai biotunnisteita. Laitteet tai resurssit usein autentikoidaan pilvipalveluun luodun sähköisen identiteetin ja sitä kautta annettujen oikeuksien tai API-avainten avulla. Toinen IAM:n päätoiminnallisuus on auktorisointi, mikä liittyy läheisesti pääsynhallintaan. Auktorisoinnin perusidea on myöntää autentikoidulle identiteetille oikeat käyttöoikeudet. Kolmas IAM:n hyöty on keskitetty hallinnointi (ks. kuvio 2). Käyttäjien ja pääsyn hallinnointi helpottuu, kun näitä voidaan hallinnoida yhdestä järjestelmästä, eikä jokaiselle käyttäjälle tarvitse esimerkiksi

tehdä tunnuksia jokaiseen palveluun ja järjestelmään erikseen. Viimeiseksi IAM helpottaa auditointia. Kun pääsynhallinta on keskitetty, voidaan helposti varmistua siitä, että tietoturvavaatimukset täyttyvät käytännössä. (Niemi n.d.)

Kuvio 2. IAM keskitetty hallinnointi (Identiteetin ja pääsynhallinta (IAM) n.d.).

3.2 Identiteetinhallinta

Identiteetinhallinnan avulla varmistutaan siitä, että kukin henkilö tai resurssi on se, joka väittää olevansa. Usein identiteetinhallinta on keskitetty yhteen palveluun, josta sitten luodaan integraatioita eri palveluihin ja järjestelmiin ja jonka kautta myös nuo ulkoiset palvelut ja järjestelmät voivat varmistua käyttäjien autentikaatiosta. Näin esimerkiksi käyttäjien ei tarvitse muistaa tunnukset kuin yhteen paikkaan, mikä helpottaa tietoturvallisuudesta huolehtimista ja heidän päivittäistä toimintaansa. Toisaalta myös tunnusten hallinnointi helpottuu, kun eri palveluihin ei tarvitse luoda omia tunnuksiaan, vaan niitä voidaan hallinnoida keskitetysti. (IAM – työntekijöille ja alihankkijoille n.d.)

Qvantel Finland Oy:n tapauksessa pääasiallinen IAM-integraatio tapahtuu HR-järjestelmän ja Active Directoryn (AD) välillä. Haluttu automatisaatio pitää huolen siitä, että AD:n identiteettitiedot ja sitä kautta tulevat sähköpostitilit yms. ovat ajantasaiset. Tähän liittyy uusien tilien luominen uusille työntekijöille, vanhojen työntekijöiden tietojen päivittäminen niiden muuttuessa, sekä poistuvien työntekijöiden tilien poistaminen. Kun tilit pidetään ajan tasalla ja keskitettynä Active Directoryyn, sitä kautta voidaan luoda yhteyksiä eri järjestelmiin, joihin käyttäjät voivat kirjautua omilla AD-tunnuksillaan Single sign-on (SSO) yhteyksien avulla.

3.3 Pääsynhallinta

Pääsynhallinta liittyy läheisesti identiteetinhallintaan, mutta ne ovat kuitenkin kaksi eri asiaa. Kun identiteetinhallinnan avulla voidaan varmistua käyttäjien tunnistuksesta, niin pääsynhallinnan avulla voidaan hallita käyttäjien pääsyä tiettyihin resursseihin, sekä sitä minkälaisia käyttöoikeuksia käyttäjillä on noissa resursseissa. Pääsynhallinnan avulla voidaan esimerkiksi määrittää, mitkä käyttäjät saavat ja voivat käyttää mitä sovelluksia ja mitä tietoja heillä on oikeus lukea tai muokata.

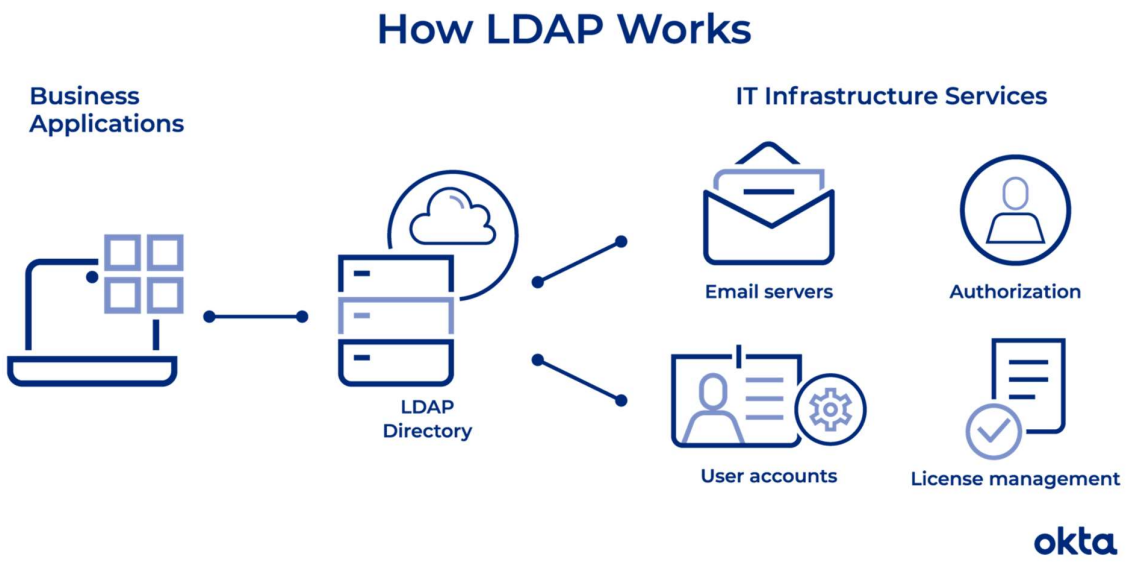
Qvantel Finland Oy:lla pääsynhallinta on useimmiten toteutettu erilaisilla Active Directory ja Azure AD rooleilla ja ryhmillä, joille on annettu erilaisia käyttöoikeuksia. Nämä ryhmät esimerkiksi joko aktivoivat SSO:n tiettyihin sovelluksiin autentikointia ja pääsyä varten tai ryhmät voidaan toimittaa eteenpäin näille sovelluksille ja niiden avulla hallinnoida tuon ulkoisen sovelluksen käyttöoikeuksia sovelluksen omilla työkaluilla.

3.4 Roolipohjainen käyttöoikeuksien hallinta

Roolipohjainen käyttöoikeuksien hallinta (RBAC eli Role-Based Access Control) on menetelmä, jolla säännellään resurssien käyttöoikeuksia yksittäisille käyttäjille annettujen roolien perusteella. RBAC:n avulla voidaan varmistaa, että käyttäjillä on pääsy vain niihin resursseihin, joihin heillä on oikeus ja että heillä on niihin resursseihin vain tarvittavat käyttöoikeudet. RBAC auttaa vähentämään tietoturvariskejä suhteessa siihen, että kaikille yksilöille asetettaisiin tarvittavat käyttöoikeudet yksittäin. Koska roolitus on systemaattista, on käyttöoikeuksia paljon helpompi hallinnoida ja inhimillisten virheiden määrä pienenee. Myös auditointi helpottuu, sillä yhdellä silmäyksellä voidaan nähdä kunkin henkilön ryhmät, tai vastaavasti yhteen ryhmään kuuluvat henkilöt. (Tunggal 2023.)

3.5 LDAP ja LDAPS

Microsoftin (What is LDAP? 2018) mukaan LDAP (Lightweight Directory Access Protocol) on avoin hakemistopalveluprotokolla, joka mahdollistaa hajautettujen hakemistopalveluiden käytön ja ylläpidon verkon yli. Hakemisto on hierarkkisesti järjestetty tietokokoelma, esimerkiksi organisaatiokaavio. LDAP:ia käytetään identiteetin- ja pääsynhallintaan. Esimerkiksi LDAP:ia voidaan käyttää käyttäjien ja tietokoneiden tunnistamiseen Windowsin verkkoympäristössä tai käyttäjätilien ja salasanojen hallintaan (ks. kuvio 3). Protokolla mahdollistaa sovelluksille nopean ja tehokkaan käyttäjätietojen kyselyn ja muokkaamisen käyttämällä standardisoituja operaatioita ja komentoja, kuten lisäys-, poisto-, etsimis- ja muokkausoperaatioita. LDAP kehitettiin alun perin yksinkertaisemmaksi vaihtoehdoksi X.500-standardin osana olevalle Directory Access Protocol (DAP) -protokollalle. LDAP perustuu asiakas-palvelinmalliin, jossa asiakas lähettää pyyntöjä palvelimelle ja vastaanottaa vastauksia. Palvelin voi olla joko omistettu LDAP-palvelin tai portti toiseen hakemistopalveluun, kuten X.500:een tai Active Directoryyn. (What Is LDAP & How Does It Work? 2023.)



Kuvio 3. LDAP-kaavio (What Is LDAP & How Does It Work? 2023).

LDAPS tarkoittaa LDAP:n käyttöä SSL/TLS-salauksen kanssa. LDAP ei salaa asiakkaan ja palvelimen välillä lähetettyjä tietoja, joten LDAPS on turvallisempi, koska se käyttää SSL/TLS-salausta. LDAP käyttää siirtoprotokollana TCP:tä ja toimii oletuksena portissa 389, kun taas LDAPS käyttää siirtoprotokollana TLS/SSL:ää ja toimii oletuksena portissa 646. TLS on uudempi versio SSL-salauksesta, joten nykyään on suositeltavaa käyttää mahdollisuuksien salliessa TLS-salausta. (LDAP vs. LDAPS: What's the Difference? 2022.)

Opinnäytetyössä LDAPS protokollaa käytettiin IAM-järjestelmän ja Active Directoryn väliseen kommunikointiin, IAM-järjestelmän ollessa asiakas ja Active Directoryn ollessa LDAP-palvelin. TLS-salausta varten Active Directory ympäristöön luotiin sertifikaatit, joita asiakassovellus voi käyttää yhteyden luomiseen. Näin voidaan luoda tietoturvallinen yhteys palvelujen välille ja tiedot kulkevat salattuina. IAM-järjestelmässä LDAPS yhteyksien hallinnointiin käytettiin Pythonin ldap3-kirjastoa (Ldap3 2020). Kirjaston avulla voidaan olio-ohjelmoinnin keinoin hallita LDAP ja LDAPS yhteyksiä ja komentoja Python-koodia käyttämällä. Näin voidaan esimerkiksi luoda uusia käyttäjiä tai muokata jo olemassa olevien käyttäjien tietoja.

4 Alusta ja ympäristö

4.1 Alusta

4.1.1 Docker ja Azure Container Registry (ACR)

Docker on avoimen lähdekoodin ohjelmisto, joka mahdollistaa sovellusten paketoinnin ja ajamisen eristetyissä konteissa. Dockerin avulla voidaan helpottaa sovellusten kehittämistä, testaamista ja jakelua eri ympäristöissä. Kontit ovat kevyitä ja nopeita käynnistää, sillä kontit eivät tarvitse kokonaista käyttöjärjestelmää, vaan käyttävät isäntäkoneen käyttöjärjestelmän resursseja. Niiden prosessit ja muut resurssit ja tiedostot ovat kuitenkin eristettyjä isäntäkoneesta ja muista konteista, mikä tekee konteista turvallisia. Kontteja voidaan käyttää eri ympäristöissä, koska ne sisältävät itsessään kaiken tiedot niiden sisältämän sovelluksen ajamiseen. Docker on siis tehokas ja monipuolinen työkalu sovellusten paketointiin ja ajamiseen konteissa. Dockerin avulla voidaan vähentää riippuvuuksia, parantaa suorituskykyä ja lisätä joustavuutta sovellusten kehityksessä ja jakelussa. (Docker overview n.d.)

Dockerin käyttöön tarvitaan ensin docker engine, joka asennetaan isäntäkoneelle. Sen jälkeen voidaan ladata tai luoda haluttuja kontteja. Konttien luomiseen käytetään dockerfile-tiedostoa, joka määrittelee kontin sisällön ja asetukset. Konttien ajamiseen käytetään docker run -komentoa, joka ottaa argumentteina kontin nimen ja mahdolliset lisäasetukset. Kontteja voidaan myös ajaa ja hallinnoida erilaisilla orkesterointityökaluilla, kuten Kuberneteksellä. (Docker overview n.d.)

Azure ACR (Azure Container Registry) on Azure pilvipalvelussa oleva palvelu, joka mahdollistaa konttikuvien rakentamisen, tallentamisen ja hallinnoinnin yksityisessä rekisterissä konttien käyttöä varten.

Kontitusteknologia sopii tehtävänantoon hyvin, sillä tehtävänannossa vaaditut järjestelmän osat on helppo jakaa loogisesti pienempiin osiin, joista jokaiselle luodaan oma kontti. Näin ollen osat ja niiden toiminnot voidaan eristää toisistaan, jolloin kehitys ja testaus on helpompaa, kun voidaan keskittyä pienempiin osa-alueisiin. Opinnäytetyön eri kontit määritettiin omilla dockerfile-tiedostoilla (ks. liite 5) ja tallennettiin Azuren ACR-palveluun, josta Kubernetes saa haettua ja ajettua tarvittavat konttikuvat.

4.1.2 Kubernetes ja Azure Managed Kubernetes Service (AKS)

Kubernetes on avoimen lähdekoodin järjestelmä, jota käytetään sovellusten käyttöönoton, skaalauksen ja hallinnan automatisointiin konttien avulla. Kubernetes ryhmittelee sovelluksen muodostavat kontit loogisiksi yksiköiksi helppoa hallintaa ja löydettävyyttä varten. Kubernetes on yksi suosituimmista ratkaisuista konttipohjaisten sovellusten kehittämiseen, toimittamiseen ja hallintaan. Kubernetes mahdollistaa sovellusten jakamisen pienempiin, itsenäisiin osiin, joita voidaan hallita erikseen. Tämä helpottaa sovellusten muokkaamista, testaamista ja päivittämistä. Kubernetes toimii useilla alustoilla pilvessä tai paikallisesti. Kubernetes optimoi resurssien käytön dynaamisesti jakamalla kuormaa konttien välillä ja skaalaamalla niitä tarpeen mukaan. Tämä säästää kustannuksia ja parantaa suorituskykyä. Kubernetes myös pystyy varmistamaan sovellusten suorittamisen häiriötilanteissa. Kubernetes pystyy havaitsemaan ja korjaamaan konttien vikoja, siirtämään niitä viallisilta solmuilta ehjille ja palauttamaan ne haluttuun tilaan. (Kubernetes Features n.d.)

Kuberneteksen pienin ja yksinkertaisin komponentti on podi, joka koostuu yhdestä tai useammasta kontista, jotka jakavat saman nimiavaruuden, tallennustilan ja elinkaaren. Podit luodaan yleensä Kuberneteksen deploymenteillä, jossa määritetään podien haluttu määrä ja tilan ylläpito. Nodet eli solmut ovat fyysisiä tai virtuaalisia koneita, jotka ajavat Kubernetes-komponentteja ja podeja. Solmuilla on asennettuna kubelet-agentti, joka kommunikoi Kubernetes API-palvelimen kanssa ja hallinnoi podin elinkaarta solmussa. Solmuilla on asennettuna myös konttien suoritukseen tarkoitettu prosessi (kuten Docker tai containerd), joka suorittaa podien kontit. (Kubernetes Features n.d.)

Azure Kubernetes Service (AKS) on Microsoft Azuren tarjoama hallittu Kubernetes-palvelu. AKS-palvelun avulla voi hyödyntää Kuberneteksen tarjoamia etuja, kuten skaalautuvuutta, joustavuutta ja tehokkuutta, ilman että tarvitsee huolehtia klusterin ylläpidosta tai infrastruktuurista. AKS:n avulla voidaan myös helpottaa Kuberneteksen pääsynhallintaa, sillä AAD-ryhmiä voidaan käyttää helposti Kuberneteksen sisäisinä ryhminä.

Opinnäytetyössä Kubernetestä käytettiin luotujen konttien ajamiseen ja ajoittamiseen. Kubernetekseen luotiin kahdelle prosessille CronJob-tyyppinen deployment, minkä avulla voitiin määrittää tarvittavat tiedot koskien kontteja, aikataulutusta, ympäristömuuttujia ja salaisuuksia varten (ks. liite 1).

4.1.3 Terraform

Terraformia käytetään infrastruktuurin hallintaan koodina (IaC – Infrastructure as Code). Terraformin avulla voidaan määritellä ja hallita pilvi-infrastruktuuria ohjelmallisesti ja automatisoidusti. Tästä on se hyöty, että muutokset voidaan dokumentoida versionhallintajärjestelmiin, ja tarvittaessa voidaan palata edellisiin versioihin. Myös laajojen ja monimutkaisten ympäristöjen monistaminen onnistuu muutamalla pienellä muutoksella Terraform-koodiin. Terraform state on tiedosto, joka sisältää Terraformin hallinnoiman infrastruktuurin tilan. Tiedosto sisältää tiedot kaikista hallinnoiduista resursseista kuten virtuaalikoneista. State-tiedoston avulla Terraform voi muutoksia tehdessä tehdä vain vähimmäismäärän tarvittavia muutoksia, eikä kaikkea tarvitse joka kerta luoda alusta asti. (Pilvi-infra haltuun koodilla - Google Cloud & Terraform 2020.)

Opinnäytetyössä Terraformia hyödynnettiin IAM-järjestelmän vaatiman Kubernetes-alustan pystyttämisen automatisoinnissa ja hallinnassa. Kubernetes-alusta on Azuren hallinnoima AKS (Azure Managed Kubernetes Service), ja alusta sekä siihen liittyvät levytilat ja salaisuuksien hallinta voidaan pystyttää Azure pilvipalveluun automaattisesti Terraformin avulla. Ensin luodaan .tf tiedosto missä määritellään tarvittavat alustat ja niiden kokomääritykset, jonka jälkeen luodaan suunnitelma ”terraform plan -out=terraform.plan” komennolla ja suunnitelman mukaiset muutokset otettiin käyttöön ”terraform apply terraform.plan" komennolla.

4.2 Active Directory

Active Directory on Microsoftin luoma tietokanta ja joukko palveluita, jotka yhdistävät käyttäjät verkon resursseihin, joita he tarvitsevat työnsä tekemiseen. Tietokanta sisältää tietoa ympäristöstä, kuten sen, mitä käyttäjiä ja tietokoneita on olemassa ja minkälaiset käyttöoikeudet kenelläkin on. Esimerkiksi tietokanta saattaa luetteloida käyttäjätilit ja niiden yksityiskohdat, kuten työnimikkeet, sähköpostiosoitteet, puhelinnumerot ja salasanat. Se tallentaa myös pääsynhallintaoikeudet. Palvelu todentaa, että jokainen käyttäjä on se, joka hän väittää olevansa, yleensä käyttäjätunnuksen, salasanan ja mahdollisesti kaksivaiheisen tunnistuksen avulla, ja sallivat käyttäjien käyttää vain niitä tietoja ja resursseja, joihin heillä on oikeus.

Active Directory yksinkertaistaa identiteetinhallintaa parantaen samalla organisaatioiden tietoturvallisuutta. AD helpottaa järjestelmänvalvojien työtä koska käyttäjien ja oikeuksien hallinta voidaan keskittää yhteen paikkaan. AD käyttää rakenteellista tietovarastoa, joka sisältää tietoja AD-objekteista, jotka ovat tyypillisesti jaettuja resursseja, kuten palvelimia, levyjä, tulostimia ja verkon käyttäjien ja tietokoneiden tilejä.

AD-objektien ja attribuuttien luokat ja niiden sisältämät tiedot määritellään joukolla sääntöjä, joita kutsutaan skeemaksi. Skeema määrittää myös objektien esiintymien rajoitukset ja rajat sekä niiden nimien muodon. Active Directory sisältää myös globaalin luettelon, joka sisältää tietoja jokaisesta hakemiston objektista. Tämän avulla käyttäjät ja järjestelmänvalvojat voivat löytää hakemistotietoja riippumatta siitä, mikä toimialue hakemistossa sisältää tiedot. Active Directory sisältää myös kysely- ja indeksointimekanismin, jotta objektit ja niiden ominaisuudet voidaan julkaista ja löytää verkon käyttäjien tai sovellusten avulla.

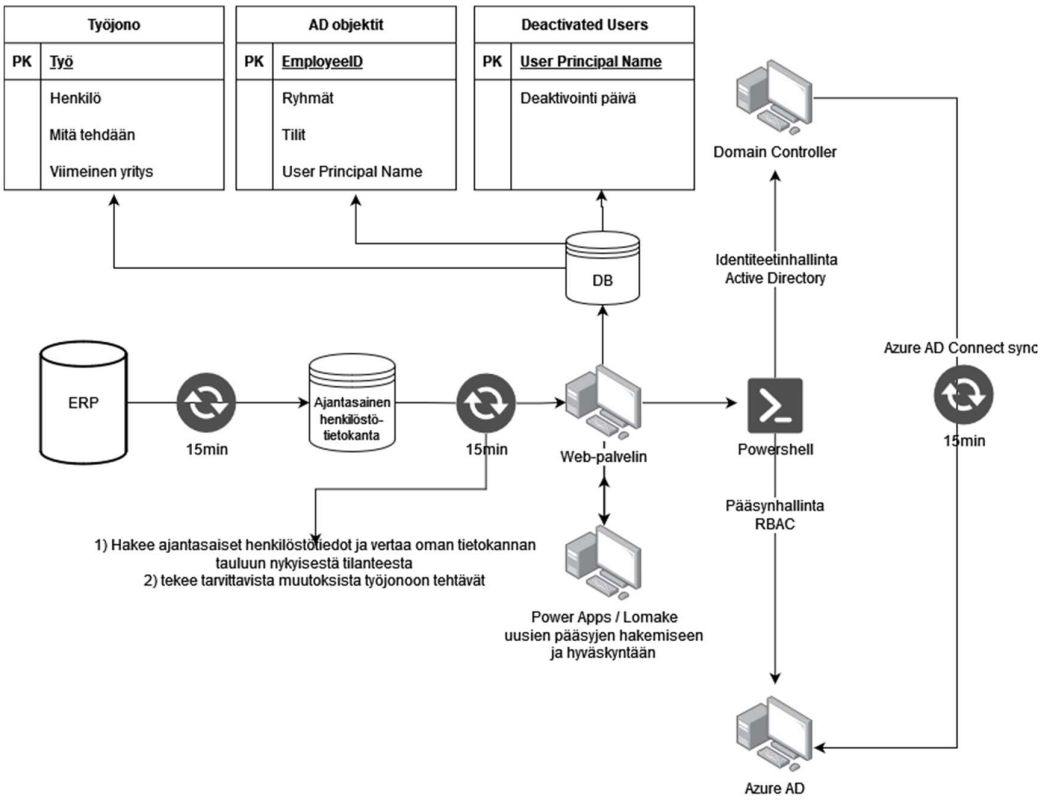
4.3 Azure AD

Azure Active Directory (Azure AD) on Microsoftin pilvipohjainen identiteetti- ja käyttöoikeusratkaisu, joka mahdollistaa organisaatioiden hallita käyttäjien ja resurssien pääsyä sekä suojata tietoja ja sovelluksia. Azure AD:n avulla voidaan toteuttaa yhtenäinen kirjautuminen (SSO) useisiin pilvisovelluksiin. AAD mahdollistaa myös roolipohjaisen käyttöoikeushallinnan (RBAC). Azure AD:ta voidaan käyttää myös itsenäisesti, mutta usein yrityksillä on ennestään laitesaleissa pyörivät perinteiset Active Directory ympäristöt ja AD:n ja AAD:n välille luodaan synkronointilinkki, jonka avulla voidaan hyödyntää molempia ympäristöjä yhtä aikaa.

5 Toteutus

5.1 Suunnittelu

Aluksi laadittiin karkea suunnitelma (ks. kuvio 4) järjestelmästä, joka täyttäisi kaikki käyttäjänhallintaan liittyvät vaatimusmäärittelyt. Ajatuksena oli, että tehdään web-palvelin, joka hakee säännöllisesti ajantasaiset henkilöstön tiedot ERP-järjestelmän tietokannasta ja vertaa tietoja IAM-järjestelmän omaan tietokantaan ja etsii mahdollisia muutoksia. Muutoksista tehdään työjonoon tehtäviä, joiden perusteella tehdään tarvittavat muutokset Active Directoryyn.



Kuvio 4. Alkuperäinen suunnitelma.

Suunniteltaessa nousi esille kaksi kohtaa, joita piti testata käytännössä ennen suunnitelman viimeistelyä, jotta löydetään paras mahdollinen toteutus työn tarpeita varten. Ensiksi Active Directoryn päivitystä varten havaittiin olevan olemassa kolme eri mahdollista teknologiaa: Power Automate, PowerShell ja suorat LDAPS-protokollan pyynnöt. Testattiin mikä näistä sopii parhaiten tähän toteutukseen. Toiseksi työjonoksi soveltuisi tietokannan taulu tai viestinvälittäjä. Testattiin kumpi sopii paremmin järjestelmän kanssa.

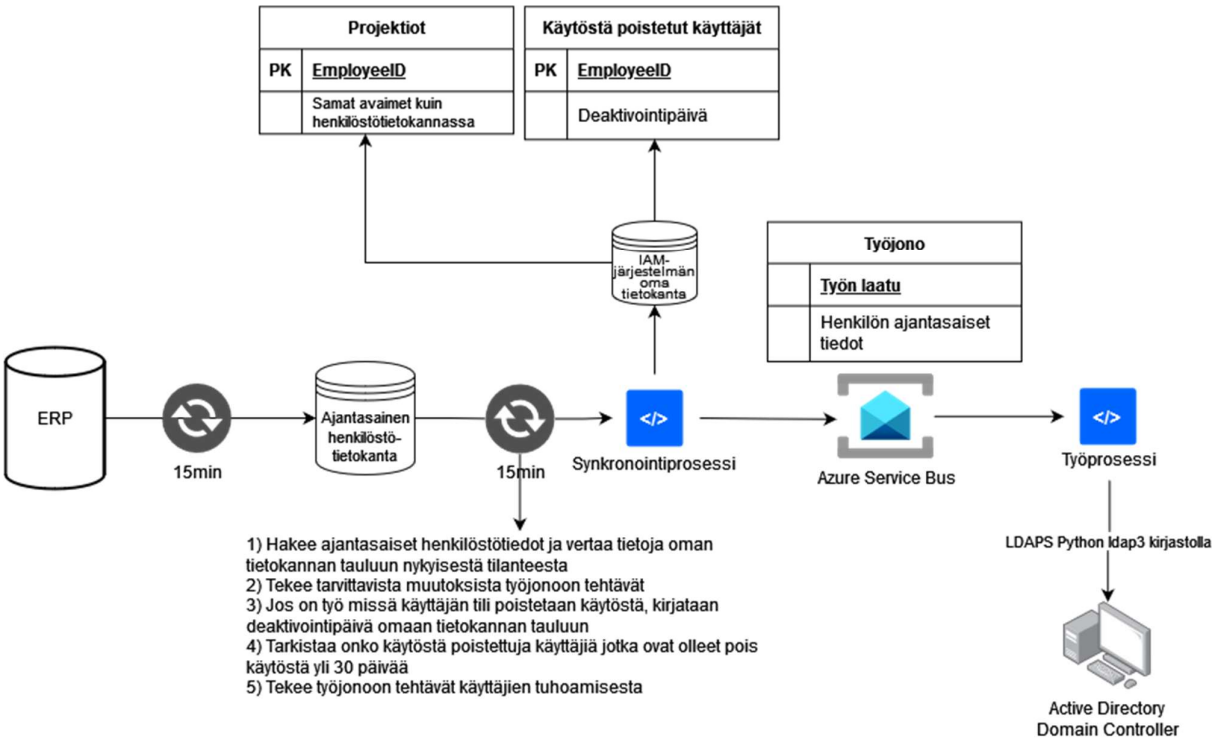
Active Directoryn päivitystä varten testattiin ensin Power Automate Desktop -sovellusta, jossa on sisäänrakennettuna valmis integraatio ja no-code työkaluja Active Directorya varten. No-code työkalujen käyttö helpottaisi järjestelmän jatkokäsittelyä sillä toteutuksesta ei tulisi niin tekninen, mutta ongelmaksi muodostui se, että Power Automaten sovellusta pitäisi ajaa aina siinä ympäristössä missä kukin Active Directory sijaitsee, ja jokainen näistä vaatisi oman lisenssinsä Microsoftilta. Tämä hajauttaisi järjestelmän osia tarpeettomasti eri ympäristöihin ja lisäisi kustannuksia, joten tästä ratkaisusta luovuttiin. Toinen mahdollisuus oli käyttää PowerShell-skriptejä, joita ajaisi suoraan web-palvelimelta. PowerShellille on olemassa valmiita moduuleja, joiden avulla voidaan

natiivisti päivittää Active Directoryn tietoja, mutta tämänkin ratkaisun osalta ongelmaksi muodostui se, että skriptejä ei voida ajaa suoraan palvelimelta, vaan ne pitäisi ajaa niissä Active Directoryn omissa ympäristöissä ja tämäkin hajauttaisi järjestelmää tarpeettomasti. Kolmatta ratkaisua, eli suoria LDAPS-protokollan mukaisia pyyntöjä varten löytyi ldap3-niminen Python-kirjasto, jonka avulla oli helppoa päivittää Active Directoryn tietoja Python-koodin avulla. Lopulta päädyttiinkin siihen ratkaisuun, että luovutaan web-palvelimesta, joka hoitaisi kaikki järjestelmän osa-alueet ja jaetaan vastuu kahdelle eri prosessille, joista toinen synkronoi tietoja ja toinen tekee ldap3-kirjaston avulla tarvittavia muutoksia.

Työjonon suhteen testattiin Azure Service Bus -viestinvälittäjää, johon synkronointiprosessi lähettää viestejä, jotka sisältävät tiedot tarvittavista muutoksista ja joita työprosessi vastaanottaa ja tekee viestien sisältämät muutokset. Koska järjestelmän eri prosessit oli suunniteltu ajettavan Azuren pilviympäristössä, oli Service Bus helppo ottaa käyttöön ja viestien välitys eri prosessien välillä oli erittäin helppokäyttöistä. Näin ollen tietokantataulun käyttöä tähän tarkoitukseen ei edes testattu, sillä viestinvälitysteknologia tuntui olevan täydellinen järjestelmän vaatimuksia varten ja uuden teknologian opettelu toisi hyvää uutta käytännön kokemusta.

Lopullisessa suunnitelmassa (ks. kuvio 5) päädyttiinkin siihen, että aluksi synkronointiprosessi vertaa 15 minuutin välein ajantasaista ERP:n päivittämää tietokantaa omaan tietokantaansa ja tekee eroavaisuuksista tehtävät, jotka se lähettää viestillä Azure Service Bus:lle. Synkronointi myös pitää huolen siitä, että sen omat projektiot vastaavat Active Directoryssa olevaa todellisuutta, jotta tarvittavien muutosten vertailu on mahdollista. Synkronointi myös tarkistaa käytöstä poistettuja henkilöitä ja luo niistä poistotehtävät, jos käyttäjät ovat olleet pois käytöstä yli 30 päivää. Työjonon käsittelyä varten suunniteltiin oma prosessi, joka vastaanottaa Azure Service Bus:sta viestejä ja tekee Active Directoryyn niissä määritellyt muutokset. Alustavasta suunnitelmasta tiputettiin pois pääsynhallinnan hakemis- ja hyväksymisprosessi, sekä Active Directoryn tietojen päivitys Azure

AD:hen. Näiden sisällyttäminen olisi paisuttanut turhaan opinnäytetyötä, sillä ne eivät liity suoranaisesti aiheeseen, mutta ovat toki hyviä jatkokehityskohteita.



Kuvio 5. Lopullinen suunnitelma.

5.2 Synkronointi

Synkronointia varten luotiin Pythonilla kirjoitettu ohjelma, joka kontitettiin ACR-repositorioon ja joka ajastettiin AKS-ympäristössä pyörivän Kuberneteksen CronJobien avulla ajamaan 15 minuutin välein. Prosessi hakee aluksi ajantasaiset tiedot henkilöstötietokannasta, ajantasaiset projektiot omasta tietokannastaan ja tiedot pois käytöstä otetuista käyttäjätileistä omasta tietokanta-taulusta. Ohjelmaan on ohjelmoitu logiikka (ks. kuvio 6, liite 6), joka käy läpi kaikki työntekijät ja vertaa työntekijän tietoja projisioituihin tietoihin. Jos näistä tiedoista löytyy eroja, logiikka tekee tarvittavat työtehtävät työprosessia varten ja päivittää omat projektionsa apufunktioiden avulla (ks. liite 7), jotta seuraava synkronointi on tietoinen muutoksista, joita varten on jo luotu vastaavat tehtävät.

```
for employee in employees:
    # check if current employee data is identical to projected data
    if employee in employee_projections:
        continue
    # if employee doesn't exist in the projected data, create a new user or projection
    if employee.qvantel_id not in projected_ids:
        if employee.ad_status in [1, 103]:  # active status
            print(create_employee(employee))
        elif employee.ad_status in [3]:  # inactive status
            print(create_projection(employee))
    # if employee exists but is modified, update the employee and projection
    else:
        print(update_employee(employee))
        if employee.ad_status in [3] and employee.qvantel_id not in disabled_ids:
            print(disable_employee(employee))
        if employee.ad_status in [1, 103] and employee.qvantel_id in disabled_ids:
            print(enable_employee(employee))
```

Kuvio 6. Synkronoinnin logiikka muutoksia varten.

Käytöstä poistettujen käyttäjätilien hallinnoimiseksi on oma logiikka (ks. kuvio 7) joka hakee tiedon siitä, milloin tili on poistettu käytöstä ja jos siitä on kulunut yli 30 päivää, luo tehtävän käyttäjätilin poistamiseksi.

```
# create a delete task for employees who have been disabled for 30 days
now = datetime.now()
month_ago = now - timedelta(days=30)

for disable in disables:
    if disable.disabled < month_ago:
        disabled_employee: Employee = next(
            employee
            for employee in employees
            if employee.qvantel_id == disable.qvantel_id
        )
        print(delete_employee(disabled_employee))
```

Kuvio 7. Synkronoinnin logiikka käytöstä poistettujen käyttäjätilien hallinnoimiseksi.

Tietokantayhteydet prosessi hoitaa Pythonin mariadb-kirjaston avulla, jota varten tarvittavat käyttäjänimet, salasanat yms. salaisuudet on tallennettuna Azuren Key Vaultiin, joista Kubernetes luo

ympäristömuuttujat. Ohjelma voi näitä ympäristömuuttujia käyttämällä luoda tarvittavat yhteydet tietokantaan. (Ks. kuvio 8.)

```
9   _DB_HOST: Final = os.getenv("DB_HOST")
10  _DB_USER: Final = os.getenv("DB_USER")
11  _DB_PASSWORD: Final = os.getenv("DB_PASSWORD")
12  _DB_NAME: Final = os.getenv("DB_NAME")
13
14
15  def connect_db() -> mariadb.Connection:
16      conn = mariadb.connect(
17          user=_DB_USER,
18          password=_DB_PASSWORD,
19          host=_DB_HOST,
20          port=3306,
21          ssl=True,
22          database=_DB_NAME,
23      )
24      return conn
```

Kuvio 8. Tietokantayhteyden luominen mariadb-kirjaston avulla.

Haut tietokannasta toteutetaan _execute_select-apufunktion ja jokaista tarvittavaa tietokannan taulua varten olevan funktion avulla (ks. kuvio 9), jotka ensin hakevat taulun tiedot listana avainarvopareja, joista luodaan ja palautetaan lista olioita. Jokainen listan olio on yksi tietokanta-taulun rivi, jossa olion attribuutit muodostuvat niin, että attribuutin avain on tietokannan otsikko ja arvo on rivin arvo kyseisen otsikon alta.

```python
def _execute_select(statement: str) -> list[dict[str, Any]]:
    with connect_db() as connection:
        with connection.cursor() as cursor:
            cursor.execute(statement)
            columns = [e[0] for e in cursor.description]
            rows = cursor.fetchall()
            return [
                {column: value for column, value in zip(columns, row)} for row in rows
            ]


def get_employees() -> list[Employee]:
    rows = _execute_select(
        "SELECT * "  # fmt: skip
        "FROM employee "
    )
    return [Employee(**e) for e in rows]
```

Kuvio 9. Haku tietokannasta.

Tietokannan päivitykseen on olemassa execute_commit-apufunktio, jolle voidaan parametrina syöttää haluttu tietokantatoiminto ja siihen liittyvät arvot. Esimerkkinä projektion lisäyksestä on funktio create_projection (ks. kuvio 10, liite 9) jossa parametreinä annetaan työntekijäolio ja funktio luo olion attribuuttien perusteella projektiotietokantatauluun uuden rivin. Tietokannoille on myös luotu omat Pythonin sisäiset dataclass-mallit, joiden avulla tietokannasta voidaan hakea tietoa ja tallentaa ne Employee-olioihin tai vaihtoehtoisesti Employee-olioista voidaan luoda tehtäviä varten json-objekteja (ks. liite 10).

```python
def execute_commit(statement: str, values: tuple | list) -> None:
    with connect_db() as connection:
        with connection.cursor() as cursor:
            cursor.execute(statement, values)
            connection.commit()


def create_projection(employee: Employee) -> str:
    execute_commit(
        "INSERT INTO employee_projections VALUES ("
        + ",".join("?" * len(asdict(employee).values()))
        + ")",
        tuple(asdict(employee).values()),
    )
    return f"Projection created for user {employee.id}"
```

Kuvio 10. Projektion luominen tietokantaan.

Viestien lähetys Azure Service Bus:lle on toteutettu Microsoftin omalla azure.servicebus Python-kirjastolla, jolla viestien lähettäminen ja vastaanottaminen on helppoa. Kuten tietokantayhteyksissä, myös viestinvälitykseen tarvittavat salaisuudet on tallennettu Azuren Key Vaultiin, josta ne haetaan ympäristömuuttujiksi yhteyden muodostamiseksi. Tehtävän luomiseksi työjonoon on funktio create_task ja apufunktio send_single_message (ks. kuvio 11, liite 8) joiden avulla lähetetään viestejä jonoon. Create_task-funktio ottaa parametrina Python sanakirjan lähetettävästä tehtävästä, joka sisältää kaksi avainarvoparia. Toinen avainarvopari sisältää tehtävän laadun ja toinen sisältää muokattavan henkilön ajantasaiset tiedot muutoksia varten.

```
9   _NAMESPACE_CONNECTION_STR: Final = os.getenv("EVENT_BUS_KEY")
10  _QUEUE_NAME: Final = os.getenv("QUEUE_NAME")
11
12
13  def send_single_message(task: dict[str, Any], sender: ServiceBusSender) -> None:
14      # Create a Service Bus message and send it to the queue
15      message = ServiceBusMessage(json.dumps(task, default=str))
16      sender.send_messages(message)
17
18
19  def create_task(task: dict[str, Any]) -> str:
20      # create a Service Bus client using the connection string
21      with ServiceBusClient.from_connection_string(
22          conn_str=_NAMESPACE_CONNECTION_STR, logging_enable=True
23      ) as servicebus_client:
24          # Get a Queue Sender object to send messages to the queue
25          sender = servicebus_client.get_queue_sender(queue_name=_QUEUE_NAME)
26          with sender:
27              # Send one message
28              send_single_message(task, sender)
29              return (
30                  f"Created {task['type']} task " f"for employee {task['employee']['id']}"
31              )
```

Kuvio 11. Funktiot Azure Service Bus -viestien lähetykseen.

5.3 Työprosessi

Työprosessi on Pythonilla kirjoitettu ohjelma, joka kontitettiin ACR-repositorioon ja joka ajastettiin AKS-ympäristössä pyörivän Kuberneteksen CronJobien avulla ajamaan 15 minuutin välein. Ohjelmaan on kirjoitettu logiikka (ks. kuvio 12, liite 2) joka hakee aluksi Azure Service Bus:sta viestit, jotka sisältävät tiedot suoritettavista töistä. Tämän jälkeen ohjelma käy viestit yksitellen läpi ja

suorittaa niissä pyydetyt muokkaukset Active Directoryyn ja muokkausten onnistuttua merkkaa viestin valmiiksi, jolloin se poistuu Service Bus:sta. Lopuksi logiikka käy läpi työntekijät, joilla ei ole esihenkilöä tallennettuna Active Directoryyn, ja päivittää tämän tiedon, jos mahdollista. Active Directoryn objektia luodessa esihenkilötieto voi jäädä tyhjäksi tapauksessa, jossa työntekijä luodaan ennen esihenkilön luomista. Vaikkakin ohjelma käyttää samoja keinoja Azure Service Bus:n yhteyden muodostamiseksi kuin synkronoinnissa, on työprosessi organisoitu eri tavalla. Tässä ohjelmassa yhteyden muodostamiseksi ei käytetä apufunktioita, vaan tehtävien läpikäyminen on sijoitettu kokonaisuudessaan ServiceBus-kontekstin alle, jotta viestit voidaan merkitä valmiiksi saman kontekstin alla missä ne on vastaanotettu.

```
def main() -> None:
    with ServiceBusClient.from_connection_string(
        conn_str=_NAMESPACE_CONNECTION_STR, logging_enable=True
    ) as servicebus_client:
        with servicebus_client.get_queue_receiver(
            _QUEUE_NAME, max_wait_time=30
        ) as receiver:
            received_msgs = receiver.receive_messages(
                max_wait_time=10, max_message_count=100
            )
            for message in received_msgs:
                task = json.loads(str(message))
                match task["type"]:
                    case "new":
                        task_result = create_user(task["employee"])
                    case "update":
                        task_result = update_user(task["employee"])
                    case "disable":
                        task_result = disable_user(task["employee"])
                    case "delete":
                        task_result = delete_user(task["employee"])
                    case "enable":
                        task_result = enable_user(task["employee"])
                    case _:  # default
                        raise ValueError(f"Unknown task type: {task['type']}")
                if task_result["result"] in [0, 68]:
                    receiver.complete_message(message)

    # populate empty manager fields
    ad_employees = get_employees_ad()
    db_employees = get_employees_db()

    for employee in ad_employees:
        if not employee["manager"]:
            db_employee = next(
                (
                    db_employee
                    for db_employee in db_employees
                    if db_employee["username"] == employee["cn"]
                ),
                None,
            )
            if db_employee:
                if db_employee["manager"]:
                    update_manager(db_employee)
```

Kuvio 12. Tehtäväjonon käsittelyn logiikka.

Työprosessissa tietokanta- ja viestinvälitysyhteydet on hoidettu samalla tavalla kuin synkronoinnissa (ks. liite 3), mutta työprosessissa uutta on LDAPS-yhteyksien luominen (ks. kuvio 13, liite 4). Jotta saadaan salattu LDAPS-yhteys pelkän LDAP-yhteyden sijaan, käytetään yhteyden luomisessa Active Directoryn Domain Controllerille tehtyjä sertifikaatteja, joiden avulla yhteys käyttää SSL/TLS-protokollia. Itse autentikaatio on hoidettu palvelutilin avulla, jonka salasana on tallennettu Azuren Key Vaultiin, ja josta taas luodaan ympäristömuuttuja ohjelman käyttöön. Huomioitavaa on, että jos IAM-järjestelmällä hoidettaisiin useampia eri ympäristöjä, tulisi connect_ldaps-funktiota muokata niin että se ottaisi parametreina halutun ympäristön tiedot, mutta nyt tähän funktioon on kovakoodattu opinnäytetyössä käytetyn testiympäristön tiedot.

```
13  def connect_ldaps() -> Connection:
14      t: Tls = Tls(
15          validate=ssl.CERT_REQUIRED,
16          local_certificate_file="../certs/prod/ldaps.pem",
17          local_private_key_file="../certs/prod/ldaps.key",
18          ca_certs_file="../certs/prod/ad_labitq.pem",
19      )
20      server: Server = Server(
21          host:"ldaps://labitq-dc1.ad.labitq", port=636, use_ssl=True, tls=t, get_info=ALL
22      )
23      conn: Connection = Connection(
24          server, user:"CN=svc_user_automation,CN=Users,DC=ad,DC=labitq", _LDAPS_PASSWORD
25      )
26      return conn
```

Kuvio 13. LDAPS-yhteyden luominen.

Kaikkia tarvittavia eri AD-toimintoja varten on ohjelmassa olemassa omat funktionsa: uuden käyttäjän luominen, käyttäjän tietojen päivitys, käyttäjän tilin poistaminen käytöstä, tilin palautus ja tilin poisto. Esimerkkinä uuden käyttäjän luominen käyttää create_user-funktiota (ks. kuvio 14) jolle annetaan parametrina käyttäjän tiedot sanakirjamuodossa avainarvopareina. Funktio kartoittaa käyttäjän eri attribuutit tietokannassa olevista arvoista Active Directoryn käyttämiin arvoihin ja muokkaa käyttäjäobjektin AD:lle sopivaan muotoon. Käyttäjäobjektin ollessa valmis funktio yrittää luoda objektin AD:hen ja jos luonti onnistuu, funktio myös määrittää tilille salasanan ja aktivoi tilin normaalina käyttäjänä. Tässä vaiheessa luotu salasana olisi mahdollista lähettää esimerkiksi tekstiviestillä uudelle työntekijälle, mutta tätä toiminnallisuutta ei ole opinnäytetyöhön sisällytetty. Jos

kaikki on mennyt toivotusti, funktio palauttaa paluuarvona tiedon siitä, ja pääohjelma voi sen perusteella merkata työn valmiiksi ja poistaa sen työjonosta.

```python
def create_user(user: dict[str, Any]) -> dict[str, Any]:
    with connect_ldaps() as conn:
        manager: str = search_manager(user["supervisor_name"])

        attributes: list[dict[str, str]] = [
            {"userPrincipalName": "email_address"},
            {"physicalDeliveryOfficeName": "site"},
            {"department": "org_team"},
            {"countryCode": "cc_num"},
            {"c": "cc_2"},
            {"telephoneNumber": "mobile_phone"},
            {"sn": "surname"},
            {"cn": "username"},
            {"co": "country_of_working"},
            {"name": "username"},
            {"facsimileTelephoneNumber": "is_manager"},
            {"employeeNumber": "qvantel_id"},
            {"title": "job_title"},
            {"pager": "employee_number"},
            {"company": "company_name"},
            {"mail": "email_address"},
            {"employeeType": "employee_type"},
        ]

        new_user = {}

        for attribute in attributes:
            key, value = list(attribute.items())[0]
            if user[value]:
                new_user[key] = user[value]

        if user["first_names"] and user["surname"]:
            new_user["displayName"] = f"{user['first_names']} {user['surname']}"

        if manager != "not found":
            new_user["manager"] = manager

        conn.add(
            dn=f'CN={user["username"]},CN=Users,DC=ad,DC=labitq',
            object_class=["top", "person", "organizationalPerson", "user"],
            new_user,
        )
        if conn.result["result"] == 0:
            password = generate_password()
            # TODO
            # Send password via SMS
            conn.extend.microsoft.modify_password(
                f'CN={user["username"]},CN=Users,DC=ad,DC=labitq', password
            )
            if conn.result["result"] == 0:
                conn.modify(
                    dn=f'CN={user["username"]},CN=Users,DC=ad,DC=labitq',
                    changes={"userAccountControl": [(MODIFY_REPLACE, [int(0x200)])]},
                )  # 0x200 normal user

        if conn.result["result"] == 0:
            print(f"Created user {user['username']}")
        return conn.result
```

Kuvio 14. Uuden käyttäjäobjektin luominen Active Directoryyn.

5.4 Active Directory -ympäristö

Active Directoryn ympäristöä varten pystytin aluksi Azureen virtuaalikoneen kokoluokassa Standard D2s v3, joka sisältää 2 vCPU:ta ja 8 Gt muistia ja käyttöjärjestelmäksi valittiin Windows Server 2022 Datacenter Azure Edition. Virtuaalikoneelle määritettiin myös virtuaalinen tietoverkko ja staattinen IP-osoite sisäverkkoon LDAPS-yhteyksiä varten. Palvelimelle luotiin lokaalit tunnukset kirjautumista varten ja kirjautumiseen käytettiin RDP-protokollaa Windowsin graafista käyttöliittymää varten.

Windows-palvelimilla on sisäänrakennettuna mahdollisuus palvella Domain Controllerina Active Directorya varten. Tämä tehdään valitsemalla palvelimelle rooli “Active Directory Domain Services” ja liittymällä jo olemassa olevaan domainiin tai luomalla uusi. Tässä tapauksessa luotiin uusi domain testiympäristöksi. Kun domain oli luotu, lisättiin palvelimelle myös rooli "Active Directory Certificate Services" sertifikaattien hallinnoimista varten LDAPS-yhteyden luomiseksi palvelimelle. Suositeltavaa on, että Certificate Authority-roolissa palvelee itsenäinen palvelin, mutta tässä testiympäristössä käytettiin Domain Controllerin ja Certificate Authorityn rooleihin yhtä palvelinta. Sertifikaattiroolin jälkeen ympäristöön luotiin juurisertifikaatti, johon kaikki palvelimen myöntämät sertifikaatit pohjautuvat, sekä sertifikaattimalli, jonka pohjalta voidaan palvelimelta hakea LDAPS-yhteyteen sopivaa sertifikaattia. Tämän jälkeen tehtiin pyyntö sertifikaatista palvelimelle, hyväksyttiin se ja tallennettiin sertifikaatti sellaiseen muotoon, jota Pythonin ldap3-kirjasto voi hyödyntää yhteyden muodostamiseksi. Näiden askeleiden jälkeen käytössä oli Active Directory hakemisto testiympäristössä, jota voitiin hallinnoida koodin avulla.

5.5 ERP-integraatio

Ympäristössä, jota opinnäytetyön IAM-järjestelmä hallinnoi oli valmiiksi olemassa MariaDB-tietokanta, jonka tauluun tallennetaan integraation avulla ERP-järjestelmästä kaikki henkilöstön ajantasaiset tiedot. Tätä samaa tietokantaa hyödynnettiin myös IAM-järjestelmässä niin, että edellä mainitusta tietokantataulusta haettiin ajantasaiset tiedot, joiden perusteella tehdä muutokset Active Directoryn hakemistoon. Tietokantaan luotiin myös kaksi uutta taulua. Ensimmäinen uusi taulu on ominaisuuksiltaan identtinen ERP-järjestelmän taulun kanssa ja se liittyi synkronointiprosessiin. Sitä käytettiin tallentamaan tiedot siitä mitkä tiedot Active Directory hakemistoon on jo tallen-

nettu ja näin voidaan helposti verrata näitä kahta taulua ja sen perusteella luoda tarvittavat tehtävät, joiden avulla Active Directoryn hakemiston tiedot saadaan ajan tasalle ERP-järjestelmään päivitettyjen tietojen kanssa. Toiseen uuteen tauluun tallennettiin tiedot pois käytöstä otetuista tileistä, ja sinne tallennettiin tilin id ja päivämäärä, milloin tili on otettu pois käytöstä. Tämän taulun avulla voidaan merkitä poistettavaksi tilejä, jotka ovat olleet pois käytöstä tarpeeksi pitkän ajan.

5.6 Työjono (Azure Service Bus)

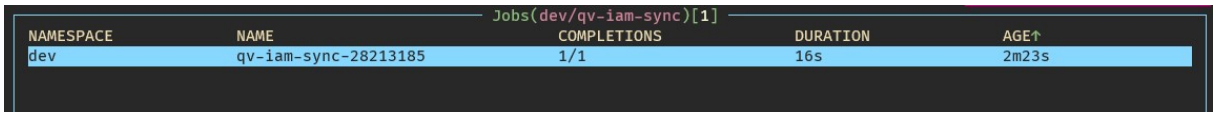
Työjonoa varten luotiin Azuren pilvipalveluun Service Bus -resurssi. Ensin luotiin Service Bus:lle nimiavaruus ja nimiavaruuteen jono. Azurella on oma azure-servicebus Python-kirjasto, jota käytettiin jonon hallinnoimiseksi. Toisesta prosessista lähetettiin jonoon tehtäviä ja toisella prosessilla luettiin jonosta tehtäviä ja merkattiin ne valmiiksi. Autentikaatio voidaan toteuttaa joko Azure AD rooleilla tai yhteysmerkkijonolla. Opinnäytetyön toteutuksessa käytettiin yksinkertaisuuden vuoksi yhteysmerkkijonoa, vaikkakin Azure AD roolitus on tietoturvallisempaa. Service Bus kirjastoa varten on olemassa Microsoftilla selkeä dokumentaatio (Send messages to and receive messages from Azure Service Bus queues (Python) 2023) jonka avulla jonon hallinnoimiseksi oli helppo tehdä koodiin funktiot.

6 Tulokset

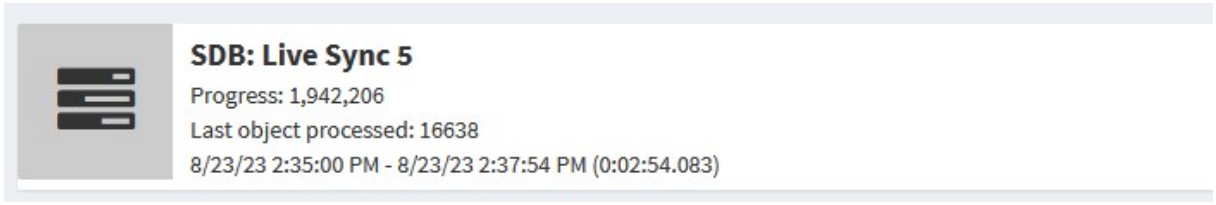
Ensimmäinen tutkimuskysymys oli, että miten luoda ERP-järjestelmän ja identiteetinhallinnan välinen automatisoitu integraatio. Tuloksena saatiin luotua järjestelmä, joka on halutusti kevyempi ja yksinkertaisempi kuin nykyinen Midpointin IAM-järjestelmä. Kehitettyä järjestelmää ajettiin testiympäristössä viikon verran, ja se sai hyvin ensin alustettua ympäristön uusilla käyttäjillä, ja myös pysyi nykyisen järjestelmän perässä päivittämällä ympäristön tietoja ajantasaisiksi. Koska opinnäytetyössä tehdyn järjestelmän tarvitsee huolehtia vain yhdenlaisista integraatioista, voidaan käyttää näille integroitaville järjestelmille natiiveja ja tehokkaampia funktioita. Opinnnäytetyössä kehitetty synkronointiprosessi sai tehtyä alustavat työjonon tehtävät 606 uudelle käyttäjälle 13 minuutissa (ks. kuvio 16) ja työprosessi sai nämä tehtävät käsiteltyä ja luotua uudet Active Directory objektit 3 minuutissa.

Kuvio 15. Kaavio Azure ServiceBus työjonosta alustavan synkronoinnin jäljiltä

Suurin hyöty kevyestä prosessista tässä tapauksessa syntyy kuitenkin rutiinisynkronointien kohdalla, sillä ilman suurempia muutoksia synkronointiprosessilla kestää n. 16 sekuntia käydä läpi tarvittavat vertailut, kun taas vanhan järjestelmän rutiinisynkronointi kestää melkein 3 minuuttia.



Kuvio 16. Synkronointiprosessiin käytettävä aika ilman tarvittavia muutoksia



Kuvio 17. Midpoint IAM järjestelmän synkronoinnin kesto

Isommassa järjestelmässä on se hyöty, että siihen voidaan helpommin integroida erilaisia ympäristöjä ja tietokantoja, mutta koska näitä mahdollisuuksia on paljon, tulee järjestelmästä helposti raskas ja opinnäytetyön kaltainen IAM-järjestelmä voi olla hyvä ratkaisu siihen, kun halutaan hallinnoida vain muutamia eri vakiintuneita ympäristöjä.

Yksi tutkimuskysymys käsitteli kysymystä, että miten voidaan varmistua siitä, että IAM-järjestelmän tiedot ovat ajantasaiset. Opinnäytetyössä tämä ongelma ratkaistiin sillä, että henkilöstötietokannasta tehtiin järjestelmälle oma kopio, johon tallennettiin projektiot siitä millainen tilanne IAM:n puolella on tällä hetkellä. Näin järjestelmän on helppo verrata ajantasaista henkilöstötietokantaa siihen minkälaisia projektioita järjestelmä on tehnyt, ja tehdä tarvittavat muutokset uudistuneen datan perusteella.

Viimeinen tutkimuskysymys liittyi siihen, että miten järjestelmästä saadaan tehtyä vikasietoinen ja kevyt. Opinnäytetyössä tämä kysymys ratkaistiin niin, että koko järjestelmä paloiteltiin kahteen eri prosessiin, jotka keskustelevat keskenään ulkoistetun työjonon kautta, joka sijaitsee pilvipalvelussa. Näin vältytään tilanteilta, joissa luullaan, että esimerkiksi joku päivitys on jo hoidettu, vaikka tuota päivitystä käsitellessä olisi tapahtunut joku vikatilanne. Synkronoinnissa päivitetään projektiot vasta kun varmistutaan siitä, että tehtävä on onnistuneesti luotu työjonoon ja työprosessi merkkaa työjonon tehtävät valmiiksi vasta kun se on varmistunut siitä, että halutut objektit ovat päivittyneet Active Directoryyn. Myös siinä tapauksessa, että prosessit kaatuisivat, ovat tarvittavat työtehtävät tallessa, sillä ne on eriytetty prosesseista ja säilytys on sysätty pilvipalvelun vastuulle. Se, että järjestelmä on jaoteltu kahteen eri prosessiin auttaa myös siinä, että prosessit pysyvät kevyinä. Laskentaresursseja ja muistia voi kulua hieman enemmän kuin yhdistetyt prosessit johtuen siitä, että prosessit eivät jaa yleisresursseja. Kuitenkin hyöty siitä, että synkronointi ja työprosessi ovat riippumattomia toisistaan, on se, että niitä voidaan ajaa vain tarvittavan ajan ja rutiininomaisessa käytössä näiden prosessien ajo pysyy nopeana ja kevyenä.

7 Pohdinta

Pääasiallinen tavoite opinnäytetyötä tehdessä oli tutustua käyttäjä- ja pääsynhallintaan ja saada syvällisempi tuntemus siitä mitä siihen liittyy ja sen tiedon perusteella luoda tavoitteiden mukainen sovellus, josta voisi olla käytännön hyötyä toimeksiantajalle. Tavoitteet täyttyivät hyvin ja tuloksena luotiin järjestelmä, josta jatkokehittelyllä voitaisiin saada korvaava järjestelmä nykyisen tilalle.

Nykyään käytetään paljon eri teknologioita, palveluita ja järjestelmiä ja erilaisten IAM-järjestelmien käyttö ja integrointi vaikuttaa erittäin hyödylliseltä sekä loppukäyttäjien, IT-tiimien sekä tietoturvan avuksi. Loppukäyttäjälle hyöty löytyy siitä, että voidaan käyttää mahdollisesti vain

yhtä sähköistä identiteettiä kaikkiin eri tarvittaviin palveluihin ja järjestelmiin, joten erilaisten käyttäjänimien ja salasanojen muistelu helpottuu. IT-tiimeille hyöty tulee siitä, että eri identiteettien ja pääsynhallinnan hallinnoiminen helpottuu, sillä näitä voidaan hallinnoida yhdestä keskitetystä paikasta. Nämä edellä mainitut hyödyt edelleen hyödyttävät tietoturvaa, sillä loppukäyttäjän käyttäessä vain yhtä sähköistä identiteettiä, voidaan esimerkiksi biometriikan ja kaksivaiheisen tunnistuksen avulla varmistua siitä, että tuo identiteetti ei pääse vääriin käsiin. Myös se, että pääsyoikeuksien hallinta on keskitettyä, helpottaa auditointia ja voidaan helpommin huomata esimerkiksi pääsyoikeuksia, joita ei pitäisi olla.

Muutamalla jatkokehityskohteella opinnäytetyössä kehitetystä järjestelmästä saataisiin vielä paljon käyttäjäystävällisempi. Suurin hyöty olisi siitä, että järjestelmälle kehitettäisiin graafinen käyttöliittymä, minkä avulla voitaisiin pysyä paremmin perillä siitä, minkälaisia tietoja IAM-järjestelmä on tallentanut ja käsitellyt. Sen avulla voitaisiin myös helpommin tehdä manuaalisia korjauksia siinä tilanteessa, että automaatio ei ole toiminut oikein.

Kehitetty järjestelmä keskittyy hyvin pitkälti identiteetinhallintaan ja toisella kehityskohteella voitaisiin laajentaa tätä roolia kattamaan myös enemmän pääsynhallintaa. Järjestelmään voitaisiin suhteellisen helposti sulauttaa ominaisuus, jossa käyttäjät voivat anoa erilaisia rooleja ja saada pääsyoikeuksia eri järjestelmiin. Näitä hakemuksia voitaisiin esimiehen hyväksynnällä käsitellä automaattisesti järjestelmän avulla. Jos näistä roolituksista tallennettaisiin tiedot myös järjestelmän omiin tietokantoihin, voidaan myös automatisoida sitä, että eri roolitukset poistetaan samalla kun käyttäjä poistetaan käytöstä, mikä lisää tietoturvaa.

Pieniä lisäyksiä myös tarvittaisiin siihen, että kehitetty järjestelmä voitaisiin ottaa tuotantokäyttöön. Nykyisessä järjestelmässä uusille työntekijöille lähetetään salasana SMS-viestillä ensimmäistä kirjautumista varten, mutta tätä ominaisuutta ei testiympäristössä haluttu testata. Toisekseen järjestelmään on kovakoodattu yhden Active Directory ympäristön tiedot, ja tuotantokäytössä tätä haluttaisiin laajentaa niin, että sama järjestelmä hoitaa tietojen päivityksen useampaan eri ympäristöön.

Lähteet

About Qvantel. 2023. Seloste Qvantelin toiminnasta omilla nettisivuilla. Viitattu 21.5.2023. https://www.qvantel.com/company/.

AWS Identity and Access Management (IAM). N.d. Tuoteseloste AWS:n sivustolla. Viitattu 12.5.2023. https://aws.amazon.com/iam/.

Docker overview. N.d. Docker dokumentaatio. Viitattu 18.4.2023. https://docs.docker.com/get-started/overview/.

IAM – keskitetty identiteetin hallinta. N.d. Mainos Telian nettisivuilla. Viitattu 18.4.2023. https://www.telia.fi/yrityksille/palvelut/luottamuspalvelut/identiteetinhallinta/.

IAM – työntekijöille ja alihankkijoille. N.d. Artikkeli Intragen Oy:n sivustolla. Viitattu 24.8.2023. https://identiteetinhallinta.fi/.

Kubernetes Features. N.d. Tuoteseloste Kuberneteksen omilla sivuilla. Viitattu 24.8.2023. https://kubernetes.io/.

Kuha. J. 2020. Pilvi-infra haltuun koodilla - Google Cloud & Terraform. Artikkeli. Viitattu 18.4.2023. https://www.nordhero.com/posts/google-cloud-terraformilla/.

Ldap3. 2020. Dokumentaatio ldap3-kirjastosta. Viitattu 28.4.2023. https://ldap3.readthedocs.io/en/latest/.

LDAP vs. LDAPS: What's the Difference? 2022. Blogikirjoitus Rublon-sivustolla. Viitattu 28.4.2023. https://rublon.com/blog/ldap-ldaps-difference/.

Niemi K. N.d. Artikkeli itewiki-sivustolla. Viitattu 12.5.2023. https://www.itewiki.fi/opas/kayttajahallinta-iam/.

Pernaa, J. 2013. Kehittämistutkimus tutkimusmenetelmänä. Kirjan luku. Viitattu 24.8.2023. http://hdl.handle.net/10138/317958/.

Pilvi-infra haltuun koodilla - Google Cloud & Terraform. 2020. Artikkeli ja opas NordHeron sivustolla. Viitattu 24.8.2023. https://www.nordhero.com/posts/google-cloud-terraformilla/.

Qvantel Finland Oy. N.d. Yritysseloste asiakastieto-sivustolla. Viitattu 21.5.2023. https://www.qvantel.com/company/.

Send messages to and receive messages from Azure Service Bus queues (Python). 2023. Dokumentaatio Microsoftin sivustolla. Viitattu 10.8.2023. https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-python-how-to-use-queues?tabs=passwordless/.

Tunggal, A. T. 2023. What is Role-Based Access Control (RBAC)? Examples, Benefits, and More. Blogikirjoitus. Viitattu 18.4.2023. https://www.upguard.com/blog/rbac/.

What is LDAP? 2018. Artikkeli Microsoftin sivustolla. Viitattu 28.4.2023. https://learn.microsoft.com/en-us/previous-versions/windows/desktop/ldap/what-is-ldap/.

What Is LDAP & How Does It Work? 2023. Artikkeli Okta-verkkosivustolla. Viitattu 28.4.2023. https://www.okta.com/identity-101/what-is-ldap/.

Liitteet

Liite 1. iam-worker/k8s_deployment.yml

```yaml
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: iam-worker-secrets
  namespace: dev
spec:
  provider: azure
  secretObjects:
    - data:
        - key: iam-worker-DB-PASSWORD
          objectName: iam-worker-DB-PASSWORD

        - key: iam-worker-EVENT-BUS-KEY
          objectName: iam-worker-EVENT-BUS-KEY

        - key: iam-worker-LDAPS-PASSWORD
          objectName: iam-worker-LDAPS-PASSWORD

      secretName: iam-worker-secrets
      type: Opaque
  parameters:
    usePodIdentity: "false"
    useVMManagedIdentity: "true"
    userAssignedIdentityID: "#####-######-#######-######"
    keyvaultName: "it-integrations"
    objects: |

      array:

        - |

          objectName: iam-worker-DB-PASSWORD
          objectType: secret
          objectVersion: ""

        - |

          objectName: iam-worker-EVENT-BUS-KEY
          objectType: secret
          objectVersion: ""

        - |

          objectName: iam-worker-LDAPS-PASSWORD
          objectType: secret
```

```
          objectVersion: ""

    tenantId: "#####-######-#######-######"
---
apiVersion: batch/v1
kind: CronJob
metadata:
  name: iam-worker
  namespace: dev
spec:
  suspend: false
  schedule: "*/15 * * * *"  # UTC
  successfulJobsHistoryLimit: 1
  failedJobsHistoryLimit: 3
  concurrencyPolicy: Forbid
  jobTemplate:
    spec:
      backoffLimit: 0
      template:
        spec:
          # Define secret volume
          volumes:
            - name: iam-worker-secrets
              csi:
                driver: secrets-store.csi.k8s.io
                readOnly: true
                volumeAttributes:
                  secretProviderClass: iam-worker-secrets
          restartPolicy: Never
          containers:
            - image: itintegrations.azurecr.io/iam-worker:0.1
              name: iam-worker
              imagePullPolicy: Always

              # Mount secrets
              volumeMounts:
                - name: iam-worker-secrets
                  mountPath: "/mnt/secrets-store"
                  readOnly: true

              env:
                # ---------- Miscellaneous ----------

                - name: APP_NAME
                  value: "iam-worker"

                # ---------- DB / Eventbus ----------

                - name: QUEUE_NAME
                  value: "iam-queue"
```

```
            - name: DB_HOST
              value: "#####-######-#######-######"

            - name: DB_USER
              value: "#####-######-#######-######"

            - name: DB_NAME
              value: "#####-######-#######-######"

            - name: DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: iam-worker-secrets
                  key: iam-worker-DB-PASSWORD

            - name: EVENT_BUS_KEY
              valueFrom:
                secretKeyRef:
                  name: iam-worker-secrets
                  key: iam-worker-EVENT-BUS-KEY

            - name: LDAPS_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: iam-worker-secrets
                  key: iam-worker-LDAPS-PASSWORD
```

Liite 2. Iam-worker/src/main.py

```python
from ldap_functions import (
    create_user,
    update_user,
    disable_user,
    delete_user,
    enable_user,
    get_employees_ad,
    update_manager,
)
from database_functions import get_employees_db
from azure.servicebus import ServiceBusClient
from typing import Final
import json
import os

_NAMESPACE_CONNECTION_STR: Final = os.getenv("EVENT_BUS_KEY")
_QUEUE_NAME: Final = os.getenv("QUEUE_NAME")


def main() -> None:
    with ServiceBusClient.from_connection_string(
        conn_str=_NAMESPACE_CONNECTION_STR, logging_enable=True
    ) as servicebus_client:
        with servicebus_client.get_queue_receiver(
            _QUEUE_NAME, max_wait_time=30
        ) as receiver:
            while True:
                received_msgs = receiver.receive_messages(
                    max_wait_time=10, max_message_count=20
                )
                if len(received_msgs) == 0:
                    break
                for message in received_msgs:
                    task = json.loads(str(message))
                    match task["type"]:
                        case "new":
                            task_result = create_user(task["employee"])
                        case "update":
                            task_result = update_user(task["employee"])
                        case "disable":
                            task_result = disable_user(task["employee"])
                        case "delete":
                            task_result = delete_user(task["employee"])
                        case "enable":
                            task_result = enable_user(task["employee"])
                        case _:  # default
                            raise ValueError(f"Unknown task type: {task['type']}")
                    if task_result["result"] in [0, 68]:
```

```
                        receiver.complete_message(message)

    # populate empty manager fields
    ad_employees = get_employees_ad()
    db_employees = get_employees_db()

    for employee in ad_employees:
        if not employee["manager"]:
            db_employee = next(
                (
                    db_employee
                    for db_employee in db_employees
                    if db_employee["username"] == employee["cn"]
                ),
                None,
            )
            if db_employee:
                if db_employee["manager"]:
                    update_manager(db_employee)


if __name__ == "__main__":
    main()
```

Liite 3. Iam-worker/src/database_functions.py

```
import os
import mariadb
from typing import Final, Any
from dotenv import load_dotenv

load_dotenv()

_DB_HOST: Final = os.getenv("DB_HOST")
_DB_USER: Final = os.getenv("DB_USER")
_DB_PASSWORD: Final = os.getenv("DB_PASSWORD")
_DB_NAME: Final = os.getenv("DB_NAME")


def connect_db() -> mariadb.Connection:
    conn = mariadb.connect(
        user=_DB_USER,
        password=_DB_PASSWORD,
        host=_DB_HOST,
        port=3306,
        ssl=True,
        database=_DB_NAME,
    )
    return conn


def _execute_select(statement: str) -> list[dict[str, Any]]:
    with connect_db() as connection:
        with connection.cursor() as cursor:
            cursor.execute(statement)
            columns = [e[0] for e in cursor.description]
            rows = cursor.fetchall()
            return [
                {column: value for column, value in zip(columns, row)} for row in
rows
            ]


def get_employees_db() -> list[dict[str, Any]]:
    results = _execute_select(
        "SELECT * "  # fmt: skip
        "FROM employee "
    )

    return results
```

Liite 4. Iam-worker/src/ldap_functions.py

```
from ldap3 import Server, Connection, Tls, ALL, MODIFY_REPLACE
from ldap3.core.exceptions import LDAPSocketOpenError
from typing import Final, Any
from support_functions import generate_password
import ssl
import os
from dotenv import load_dotenv
from time import sleep

load_dotenv()

_LDAPS_PASSWORD: Final = os.getenv("LDAPS_PASSWORD")


def connect_ldaps() -> Connection:
    t: Tls = Tls(
        validate=ssl.CERT_REQUIRED,
        local_certificate_file="certs/prod/ldaps.pem",
        local_private_key_file="certs/prod/ldaps.key",
        ca_certs_file="certs/prod/ad_labitq.pem",
    )
    server: Server = Server(
        "ldaps://labitq-dc1.ad.labitq", port=636, use_ssl=True, tls=t, get_info=ALL
    )
    for i in range(0, 4):
        try:
            conn: Connection = Connection(
                server,
                "CN=svc_user_automation,CN=Users,DC=ad,DC=labitq",
                _LDAPS_PASSWORD,
            )
            return conn
        except LDAPSocketOpenError:
            print(f"Connection failed, fails: {i+1}")
            sleep(2)
            continue
    raise ConnectionError


def get_employees_ad() -> list[dict[str, Any]]:
    with connect_ldaps() as conn:
        conn.search(
            "DC=ad,DC=labitq", "(objectclass=person)", attributes=["cn", "manager"]
        )
        results: list[dict[str, Any]] = conn.entries
        return results
```

```python
def search_manager(manager: str) -> str:
    if not manager:
        return "not found"
    if "non-active" in manager:
        return "not found"
    with connect_ldaps() as conn:
        conn.search(
            "DC=ad,DC=labitq",
            f"(displayName={manager})",
            attributes=["distinguishedName"],
        )
        try:
            entry = conn.entries[0]
            distinguished_name = entry.entry_to_json()["attributes"][
                "distinguishedName"
            ][0]
            return distinguished_name
        except (KeyError, IndexError, TypeError):
            return "not found"


def create_user(user: dict[str, Any]) -> dict[str, Any]:
    with connect_ldaps() as conn:
        manager: str = search_manager(user["supervisor_name"])

        attributes: list[dict[str, str]] = [
            {"userPrincipalName": "email_address"},
            {"physicalDeliveryOfficeName": "site"},
            {"department": "org_team"},
            {"countryCode": "cc_num"},
            {"c": "cc_2"},
            {"telephoneNumber": "mobile_phone"},
            {"sn": "surname"},
            {"cn": "username"},
            {"co": "country_of_working"},
            {"name": "username"},
            {"facsimileTelephoneNumber": "is_manager"},
            {"employeeNumber": "qvantel_id"},
            {"title": "job_title"},
            {"pager": "employee_number"},
            {"company": "company_name"},
            {"mail": "email_address"},
            {"employeeType": "employee_type"},
        ]

        new_user = {}

        for attribute in attributes:
            key, value = list(attribute.items())[0]
            if user[value]:
```

```
                new_user[key] = user[value]

        if user["first_names"] and user["surname"]:
            new_user["displayName"] = f"{user['first_names']} {user['surname']}"

        if manager != "not found":
            new_user["manager"] = manager

        conn.add(
            f'CN={user["username"]},CN=Users,DC=ad,DC=labitq',
            ["top", "person", "organizationalPerson", "user"],
            new_user,
        )
        if conn.result["result"] == 0:
            password = generate_password()
            # TODO
            # Send password via SMS
            conn.extend.microsoft.modify_password(
                f'CN={user["username"]},CN=Users,DC=ad,DC=labitq', password
            )
            if conn.result["result"] == 0:
                conn.modify(
                    f'CN={user["username"]},CN=Users,DC=ad,DC=labitq',
                    {"userAccountControl": [(MODIFY_REPLACE, [int(0x200)])]},
                )  # 0x200 normal user

        if conn.result["result"] == 0:
            print(f"Created user {user['username']}")
        return conn.result


def update_user(user: dict[str, Any]) -> dict[str, Any]:
    with connect_ldaps() as conn:
        manager = search_manager(user["supervisor_name"])
        if manager == "not found":
            manager = ""

        attributes: list[dict[str, str]] = [
            {"physicalDeliveryOfficeName": "site"},
            {"department": "org_team"},
            {"countryCode": "cc_num"},
            {"c": "cc_2"},
            {"telephoneNumber": "mobile_phone"},
            {"sn": "surname"},
            {"co": "country_of_working"},
            {"facsimileTelephoneNumber": "is_manager"},
            {"employeeNumber": "qvantel_id"},
            {"title": "job_title"},
            {"pager": "employee_number"},
            {"company": "company_name"},
```

```
            {"employeeType": "employee_type"},
        ]

        updated_user: dict[str, Any] = {}

        for attribute in attributes:
            key, value = list(attribute.items())[0]
            if user[value]:
                updated_user[key] = [(MODIFY_REPLACE, [user[value]])]

        if user["first_names"] and user["surname"]:
            display_name = f"{user['first_names']} {user['surname']}"
            updated_user["displayName"] = [(MODIFY_REPLACE, [display_name])]

        if manager:
            updated_user["manager"] = [(MODIFY_REPLACE, [manager])]

        conn.modify(f'CN={user["username"]},CN=Users,DC=ad,DC=labitq', up-
dated_user)
        if conn.result["result"] == 0:
            print(f"Updated user {user['username']}")
        return conn.result


def update_manager(user: dict[str, Any]) -> dict[str, Any]:
    with connect_ldaps() as conn:
        manager = search_manager(user["supervisor_name"])
        if manager == "not found":
            manager = ""
        conn.modify(
            f'CN={user["username"]},CN=Users,DC=ad,DC=labitq',
            {
                "manager": [(MODIFY_REPLACE, [manager])],
            },
        )
        return conn.result


def disable_user(user: dict[str, Any]) -> dict[str, Any]:
    with connect_ldaps() as conn:
        conn.modify(
            f'CN={user["username"]},CN=Users,DC=ad,DC=labitq',
            {"userAccountControl": [(MODIFY_REPLACE, [int(0x202)])]},
        )
        result = conn.result
        if result["result"] == 0:
            print(f"Disabled user {user['username']}")
        conn.unbind()
        return result
```

```
def enable_user(user: dict[str, Any]) -> dict[str, Any]:
    with connect_ldaps() as conn:
        conn.modify(
            f'CN={user["username"]},CN=Users,DC=ad,DC=labitq',
            {"userAccountControl": [(MODIFY_REPLACE, [int(0x200)])]},
        )
        if conn.result["result"] == 0:
            print(f"Enabled user {user['username']}")
        return conn.result


def delete_user(user: dict[str, Any]) -> dict[str, Any]:
    with connect_ldaps() as conn:
        conn.delete(f'CN={user["username"]},CN=Users,DC=ad,DC=labitq')
        if conn.result["result"] == 0:
            print(f"Deleted user {user['username']}")
        return conn.result
```

Liite 5. Iam-sync/Dockerfile

```
FROM python:3.11.3-buster

ENV LD_PRELOAD=/usr/lib/mariadb/libmariadb.so

RUN apt-get update && apt-get install -y gnupg2 && apt-get -y install curl && apt-
get  -y install gcc

RUN wget https://dlm.mariadb.com/2678579/Connectors/c/connector-c-3.3.3/mariadb-
connector-c-3.3.3-debian-buster-amd64.tar.gz -O - | tar -zxf - --strip-components=1
-C /usr

# Create App directory
ARG APP_PATH=/qv-iam-sync
RUN mkdir ${APP_PATH} \
 && mkdir ${APP_PATH}/certs

# Copy application files
COPY src/ ${APP_PATH}
COPY certs/prod/*.key ${APP_PATH}/certs/
COPY certs/prod/*.pem ${APP_PATH}/certs/
COPY requirements.txt ${APP_PATH}/

RUN pip install --no-cache-dir -U pip setuptools \
 && pip install --no-cache-dir -r ${APP_PATH}/requirements.txt

WORKDIR ${APP_PATH}

CMD ["python", "main.py"]
```

Liite 6. Iam-sync/src/main.py

```
from datetime import datetime, timedelta
from functions import (
    create_employee,
    create_projection,
    update_employee,
    disable_employee,
    enable_employee,
    delete_employee,
)
from database.client import get_employees, get_projections, get_disables
from database.models import Employee


def main() -> None:
    employees = get_employees()
    employee_projections = get_projections()
    disables = get_disables()

    projected_ids: list[int] = [
        employee.id for employee in employee_projections
    ]
    disabled_ids: list[int] = [disable.id for disable in disables]

    for employee in employees:
        # check if current employee data is identical to projected data
        if employee in employee_projections:
            continue
        # if employee doesn't exist in the projected data, create a new user or
projection
        if employee.id not in projected_ids:
            if employee.ad_status in [1, 103]:  # active status
                print(create_employee(employee))
            elif employee.ad_status in [3]:  # inactive status
                print(create_projection(employee))
        # if employee exists but is modified, update the employee and projection
        else:
            print(update_employee(employee))
            if employee.ad_status in [3] and employee.id not in disabled_ids:
                print(disable_employee(employee))
            if employee.ad_status in [1, 103] and employee.id in disabled_ids:
                print(enable_employee(employee))

    # create a delete task for employees who have been disabled for 30 days
    now = datetime.now()
    month_ago = now - timedelta(days=30)

    for disable in disables:
        if disable.disabled < month_ago:
```

```
            disabled_employee: Employee = next(
                employee
                for employee in employees
                if employee.id == disable.id
            )
            print(delete_employee(disabled_employee))


if __name__ == "__main__":
    main()
```

Liite 7. Iam-sync/src/fucntions.py

```
from typing import Any
from dataclasses import asdict
from database.models import Employee
from database.client import execute_commit
from servicebus.client import create_task


def create_employee(employee: Employee) -> str:
    task: dict[str, Any] = {
        "type": "new",
        "employee": asdict(employee),
    }
    execute_commit(
        "INSERT INTO employee_projections VALUES ("
        + ",".join("?" * len(asdict(employee).values()))
        + ")",
        tuple(asdict(employee).values()),
    )
    create_task(task)
    return f"Projection created for new user {employee.id}"


def create_projection(employee: Employee) -> str:
    execute_commit(
        "INSERT INTO employee_projections VALUES ("
        + ",".join("?" * len(asdict(employee).values()))
        + ")",
        tuple(asdict(employee).values()),
    )
    return f"Projection created for user {employee.id}"


def update_employee(employee: Employee) -> str:
    task: dict[str, Any] = {
        "type": "update",
        "employee": asdict(employee),
    }
    execute_commit(
        "REPLACE INTO employee_projections VALUES ("
        + ",".join("?" * len(asdict(employee).values()))
        + ")",
        tuple(asdict(employee).values()),
    )
    create_task(task)
    return f"Projection updated for user {employee.id}"


def enable_employee(employee: Employee) -> str:
```

```
    task: dict[str, Any] = {
        "type": "enable",
        "employee": asdict(employee),
    }
    execute_commit(
        "DELETE FROM disabled_employees WHERE qvantel_id = ?", [employee.id]
    )
    create_task(task)
    return f"Enabled user {employee.id}"


def disable_employee(employee: Employee) -> str:
    task: dict[str, Any] = {
        "type": "disable",
        "employee": asdict(employee),
    }
    execute_commit(
        "INSERT INTO disabled_employees VALUES (?, current_timestamp())",
        [employee.id],
    )
    create_task(task)
    return f"Disabled user {employee.id}"


def delete_employee(employee: Employee) -> str:
    task: dict[str, Any] = {
        "type": "delete",
        "employee": asdict(employee),
    }
    execute_commit(
        "DELETE FROM disabled_employees WHERE id = ?", [id]
    )
    create_task(task)
    return f"Deleted user {employee.id}"
```

Liite 8. Iam-sync/src/servicebus/client.py

```python
from azure.servicebus import ServiceBusMessage, ServiceBusClient, ServiceBusSender
from typing import Final, Any
import json
import os
from dotenv import load_dotenv

load_dotenv()

_NAMESPACE_CONNECTION_STR: Final = os.getenv("EVENT_BUS_KEY")
_QUEUE_NAME: Final = os.getenv("QUEUE_NAME")


def send_single_message(task: dict[str, Any], sender: ServiceBusSender) -> None:
    # Create a Service Bus message and send it to the queue
    message = ServiceBusMessage(json.dumps(task, default=str))
    sender.send_messages(message)


def create_task(task: dict[str, Any]) -> str:
    # create a Service Bus client using the connection string
    with ServiceBusClient.from_connection_string(
        conn_str=_NAMESPACE_CONNECTION_STR, logging_enable=True
    ) as servicebus_client:
        # Get a Queue Sender object to send messages to the queue
        sender = servicebus_client.get_queue_sender(queue_name=_QUEUE_NAME)
        with sender:
            # Send one message
            send_single_message(task, sender)
            return (
                f"Created {task['type']} task "
                f"for employee {task['employee']['id']}"
            )
```

Liite 9. Iam-sync/src/database/client.py

```python
import os
import mariadb
from typing import Final, Any
from dotenv import load_dotenv
from database.models import Employee, Disable

load_dotenv()

_DB_HOST: Final = os.getenv("DB_HOST")
_DB_USER: Final = os.getenv("DB_USER")
_DB_PASSWORD: Final = os.getenv("DB_PASSWORD")
_DB_NAME: Final = os.getenv("DB_NAME")


def connect_db() -> mariadb.Connection:
    conn = mariadb.connect(
        user=_DB_USER,
        password=_DB_PASSWORD,
        host=_DB_HOST,
        port=3306,
        ssl=True,
        database=_DB_NAME,
    )
    return conn


def _execute_select(statement: str) -> list[dict[str, Any]]:
    with connect_db() as connection:
        with connection.cursor() as cursor:
            cursor.execute(statement)
            columns = [e[0] for e in cursor.description]
            rows = cursor.fetchall()
            return [
                {column: value for column, value in zip(columns, row)} for row in
rows
            ]


def get_employees() -> list[Employee]:
    rows = _execute_select(
        "SELECT * "  # fmt: skip
        "FROM employee "
    )
    return [Employee(**e) for e in rows]


def execute_commit(statement: str, values: tuple | list) -> None:
    with connect_db() as connection:
```

```
        with connection.cursor() as cursor:
            cursor.execute(statement, values)
            connection.commit()


def get_projections() -> list[Employee]:
    rows = _execute_select(
        "SELECT * "  # fmt: skip
        "FROM employee_projections ",
    )
    return [Employee(**e) for e in rows]


def get_disables() -> list[Disable]:
    rows = _execute_select(
        "SELECT * "  # fmt: skip
        "FROM disabled_employees ",
    )
    return [Disable(**e) for e in rows]
```

Liite 10. Iam-sync/src/database/models.py

```
from dataclasses import dataclass, asdict
from datetime import datetime
import json


@dataclass()
class Disable:
    id: int
    disabled: datetime


@dataclass()
class Employee:
    source: str
    id: int
    employee_type: str
    role_type: str
    employee_number: int
    new_employee_number: int
    old_employee_number: int
    username: str
    preferred_username: str
    sympa_username: str
    operational_ad: str
    supervisor: int
    supervisor_name: str
    surname: str
    surname_copy: str
    first_names: str
    preferred_surname: str
    preferred_given_name: str
    sympa_email_address: str
    email_address: str
    other_email_address: str
    mobile_phone: str
    job_title: str
    job_title_standard: str
    country_id: int
    country_of_working: str
    cc_2: str
    cc_3: str
    cc_num: str
    company_id: int
    company_name: str
    org_unit_1: str
    org_unit_1_name: str
    org_unit_2: str
    org_unit_2_name: str
```

```
    org_unit_3: str
    org_unit_3_name: str
    org_unit_4: str
    org_unit_4_name: str
    org_unit_5: str
    org_unit_5_name: str
    cost_center_id: int
    cost_center: str
    cost_center_1: str
    cost_center_2: str
    org_team: str
    ad_status: int
    ad_status_name: str
    modified: datetime
    start_of_employment: datetime
    start_of_agreement: datetime
    end_of_agreement: datetime
    worktime: float
    site_id: int
    site: str
    status: str
    message: str
    ad_email_address: str
    ad_email_aliases: str
    vendor_company_id: int
    vendor_company: str
    company_credit_card: int
    collective_agreement: str
    is_manager: str
    manager: str
    last_modify: datetime

    def as_json(self) -> str:
        return json.dumps(asdict(self), default=str)
```