SeAMK

SEINÄJOEN AMMATTIKORKEAKOULU
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Pasi Palomäki

IoT-järjestelmän suunnittelu ja toteutus

Opinnäytetyö
Syksy 2023
Insinööri (ylempi AMK), Teknologiaosaamisen johtaminen

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Tutkinto-ohjelma: Insinööri (ylempi AMK), Teknologiaosaamisen johtaminen

Tekijä: Pasi Palomäki

Työn nimi: IoT-järjestelmän suunnittelu ja toteutus

Ohjaaja: Raine Kauppinen

Vuosi: 2023 Sivumäärä:66 Liitteiden lukumäärä:

Opinnäytetyössä tutkittiin IoT-järjestelmän suunnittelua ja toteutusta KajaPro Oy:lle. KajaPro Oy on Kajaanissa sijaitseva IT-yritys, joka tarjoaa ohjelmistotuotannon alihankintapalveluja ohjelmistojen kehittämiseen. Opinnäytetyön aihe syntyi, kun tahdottiin toteuttaa IoT-järjestelmä koe- ja opetuskäyttöön. Työn tarkoituksena oli suunnitella toimiva arkkitehtuurimalli IoT-laitteille ja pilvipalveluille, jotta on mahdollista korvata pilvipalvelut omilla palvelimilla.

Arkkitehtuurimallin suunnittelua lähdettiin tutkimaan käyttäen konstruktiivista tutkimusmenetelmää. Suunnittelu aloitettiin perehtymällä kahteen eri arkkitehtuurimalliin, joiden ideologioita käytettiin soveltaen järjestelmän arkkitehtuurisuunnittelussa, jolle oli vaatimusmäärittelyssä asetettu rajoitteita sekä esivalittu ESP32-pohjainen mikrokontrolleri IoT-laitteeksi. Tietoa kerättiin internetistä, kirjoista ja artikkeleista. Näiden pohjalta lähdettiin toteuttamaan järjestelmän suunnittelua.

Suunnittelun toteutuksessa yhdistettiin IoT-laite pilvessä olevaan palveluun ja ohjelmoitiin palvelinohjelma. Palvelinohjelma mahdollisti verkkosivuna internetissä olevalta käyttöliittymältä käskyjen lähettämisen IoT-laitteelle. Tässä oli käytetty hyödyksi REST API -kutsuja ja MQTT-protokollaa. Näiden kahden eri kutsujen kohtaamispisteenä toimi palvelinohjelma, joka oli pystytettynä Azure-pilvipalvelussa. Pilvipalveluissa olevista palveluista saatiin tehtyä hyvä dokumentointi ja ne olivat skaalattavissa tarpeen mukaan.

Opinnäytetyössä saatiin suunniteltua järjestelmä, joka vastasi toimeksiantajan vaatimuksia. Suunnitellun järjestelmän toteutus onnistui vastaamaan näitä vaatimuksia ja samalla saatiin hyvin dokumentoitu toteutus, jossa on käytetty moderneja ohjelmistokehitysperiaatteita sekä automatisoitu testausta.

[1] Asiasanat: konstruktiivinen tutkimus, järjestelmäarkkitehtuuri, arkkitehtuurimalli, pilvipalvelut, IoT

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Degree programme: Master of Engineering, Technology Competence Management

Author: Pasi Palomäki

Title of thesis: IoT system design and implementation

Supervisor: Raine Kauppinen

Year: 2023 Number of pages:66 Number of appendices:

The thesis researched the design and implementation of an IoT system for KajaPro Oy. KajaPro is an IT company located in Kajaani offering software production subcontracting services for software development. The subject of the thesis arose out of a need when the company wanted to implement an IoT system for test and teaching use. The purpose of the thesis was to design a functional architecture model for IoT devices and cloud services where it was possible to replace cloud services with own servers.

The research was initiated using the constructive research method. The design commenced by studying two distinct architectural models which were incorporated into the system's design. The system's design had predefined constraints outlined in the requirement specification. ESP32-based microcontroller was selected as the IoT device. Information was gathered from a variety of sources which included the internet, books, and articles. Implementation strategy was devised that served as the foundation for planning.

During the implementation connectivity between the IoT device and a cloud-based service was established while developing the server program. Server program facilitated user commands via a web-based interface. Used REST API calls and the MQTT protocol converging at the server program hosted on the Azure cloud service platform. These cloud services were not only well-documented but also scalable according to the needs.

The thesis work resulted in a system that successfully met the client's requirements. The execution of the planned system not only satisfied these prerequisites but also produced a documented implementation adhering to modern software development principles and incorporating automated testing.

[1] Keywords: constructive research, system architecture, architectural model, cloud services, IoT

SISÄLTÖ

Opinnäytetyön tiivistelmä 2
Thesis abstract 3
SISÄLTÖ 4
Kuva- ja kuvioluettelo 6
Käytetyt termit ja lyhenteet 7
1 JOHDANTO 9
1.1 Työn tausta ja tavoite 9
1.2 Kehittämismenetelmä 11
2 JÄRJESTELMÄARKKITEHTUURI 13
2.1 Tarkoitus 13
2.2 Järjestelmäarkkitehtuurin periaatteet 14
2.3 Ylhäältä alas -lähestymistapa järjestelmäarkkitehtuurin suunnittelussa 16
2.4 Alhaalta ylös -lähestymistapa järjestelmäarkkitehtuurin suunnittelussa 17
2.5 Monoliittinen järjestelmäarkkitehtuurimalli 18
2.6 Mikropalvelut 19
2.7 Tietoturva arkkitehtuurin osana 23
2.8 Dokumentointi 24
3 IoT – ESINEIDEN INTERNET 27
3.1 Yleisesti esineiden internet 27
3.2 Langaton tiedonsiirto WiFillä 27
3.3 TCP/IP-tiedonsiirtoprotokolla 28
3.4 Verkkopalvelu 30
3.4.1 SOAP & REST verkkopalveluissa 30
3.4.2 Verkkopalvelun valvonta ja vikasietoisuus 31
3.5 MQTT-viestintä 32
3.6 Mikrokontrolleri ja mittalaitteet 34
3.7 Haavoittuvuudet ja niiden estäminen 35
3.8 Käyttäjien ja laitteiden hallinta 36

3.9 OpenSSL-salaus 37
3.10 Azuren pilvipalvelu alustana palvelulle 38
4 PROTOTYYPIN JÄRJESTELMÄARKKITEHTUURI 40
4.1 Laitekerros 40
4.2 Mikropalvelukerros 41
4.2.1 REST API ja MQTT-kommunikointi 42
4.2.2 Virtuaalipalvelin ja pilvipalvelu 43
4.2.3 Mikropalvelut Docker-imagena ja pilvipalvelussa 45
4.2.4 Tietokantapalvelu 46
4.3 Käyttöliittymäkerros 46
4.4 Dokumentaation suunnittelu & toteutus 47
4.5 Tietoturvan suunnittelu, valvonta ja testaus 48
5 PROTOTYYPIN TOTEUTUS JA TULOKSET 50
5.1 Mikrokontrolleri ja sensori 50
5.2 Verkkopalvelimen toteutus 52
5.3 Käyttöliittymä ja tunnistautuminen 53
5.4 Dokumentaatio ja versiohallinta 55
5.5 Tulokset ja havainnot 57
6 JOHTOPÄÄTÖKSET JA POHDINTA 62
LÄHTEET 65

Kuva- ja kuvioluettelo

Kuva 12. IoT-laitteen graafinen näkymä. 51

Kuva 13. MQTT-viestit ja laitteen debuggaus sarjaportista. 52

Kuva 14. API-kutsujen ja MQTT-käskyjen debuggaus. 53

Kuva 15. Käyttöliittymän ulkoasu. 54

Kuva 16. Käyttöliittymään kirjautuminen. 55

Kuva 17. Esimerkki dokumentin visualisoinnista. 56

Kuvio 1. IoT-järjestelmän rakenne. 9

Kuvio 2. Konstruktiivisen tutkimuksen prosessi. 12

Kuvio 3. Esimerkki monoliittisesta arkkitehtuurista. 19

Kuvio 4. Mikropalvelupohjaisen järjestelmäarkkitehtuurin rakenne. 20

Kuvio 5. Erilaisten mikropalveluiden yhdistäminen. 21

Kuvio 6. Ohjelmistoarkkitehtuurin 4+1-näkymämalli. 25

Kuvio 7. Esimerkki verkkopalvelusta. 30

Kuvio 8. API-kuorman jako verkkopalvelussa. 32

Kuvio 9. MQTT-protokollan rakenne. 34

Kuvio 10. IoT-Datan keräys ja siirto. 40

Kuvio 11. Esimerkki MQTT-viestiliikenteestä ja REST API-komennoista. 43

Kuvio 18. IoT-järjestelmän toteutunut rakenne. 58

Käytetyt termit ja lyhenteet

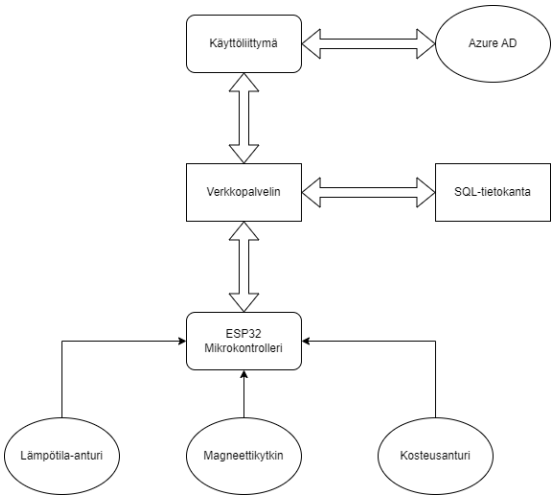
API	Application Programming Interface on standardin mukainen yhtymäkohta, joka mahdollistaa tietojen noutamisen ohjelmien välillä.
Arduino IDE	Avoimen lähdekoodin ohjelmisto, jota käytetään koodin kirjoittamiseen ja lataamiseen mikrokontrollerille.
Azure AD	Pilvipalvelu, joka tarjoaa järjestelmänvalvojille mahdollisuuden hallita käyttäjiä ja heidän käyttöoikeuksiaan.
Azure DevOps	Microsoft-tuote, joka tarjoaa versionhallinnan, vaatimusten hallinnan, raportoinnin, laboratoriohallinnan, projektinhallinnan, testauksen, automatisoidut koontiversiot ja julkaisujen hallinta-ominaisuudet. Tämä kattaa koko elinkaarisovelluksen ja mahdollistaa DevOps-ominaisuudet.
Debug	Ohjelmistokehitystermi virheenkorjaukselle tai virheiden etsimisille ohjelmakoodista.
Docker	Docker on työkalu, joka on suunniteltu helpottamaan sovellusten luomista, käyttöönottoa ja suorittamista.
GDPR	General Data Protection Regulation, EU:n yleinen tietosuoja-asetus.
IoT	Internet of things, esineiden internet.
JSON	JavaScript Object Notation, yksinkertainen ja kevyt avoimen standardin tiedostomuoto tiedonvälitykseen ja tallennukseen.
MQTT	Message Queuing Telemetry Transport, viestintäprotokolla.
PaaS	Platform as a Service, tarkoittaa palvelualustan ulkoistamista, tyypillisesti pilvipalveluna.

SSO	Single Sign-On, Kertakirjautuminen on menetelmä, jossa pääsy useisiin palveluihin toteutetaan yhdellä käyttäjän todennuksella.
URI	Uniform Resource Identifier on merkkijono, jolla kerrotaan tietyn tiedon paikka tai yksikäsitteinen nimi.
XML	Extensible Markup Language, merkintäkielien standardi, joka määrittää tietojen merkintämuodon loogisella rakenteella.

1 JOHDANTO

1.1 Työn tausta ja tavoite

Opinnäytetyön tarkoituksena on toteuttaa Kajapro Oy:lle IoT-järjestelmän prototyyppi, jota voitaisiin käyttää sisäisessä kehityksessä ja opetuksessa. Järjestelmän tarkoituksena on mitata anturien mittaustuloksia. Järjestelmää ohjataan verkkosovelluksesta, jossa on nähtävissä antureista saatu mittaustieto. Järjestelmä jakaantuu pääpiirteissään mikrokontrolleriin, verkkopalvelimeen ja käyttöliittymään. Käyttöliittymässä on käyttäjän tunnistus, johon käyttäjä kirjautuu sisään. Sisäänkirjautumisen jälkeen käyttäjä pääsee lukemaan antureilta tallennettua tietoa, joka näytetään verkkosivulla. Anturien lähettämä tieto noudetaan verkkopalvelimella olevasta tietokannasta. Verkkopalvelimella on oma palvelu, joka hoitaa viestikommunikaatiota ja tiedon tallennusta käyttöliittymän, tietokannan ja mittaustuloksia lähettävän mikrokontrollerin välillä. IoT-järjestelmän arkkitehtuuri suunnitellaan ja toteutetaan mahdollisimman modulaariseksi, jotta siinä olevia osia pystytään käyttämään uudelleen ohjelmistokirjastoina tai omina kokonaisuuksinaan. Järjestelmän suunniteltu rakenne käy ilmi kuviosta 1.



Kuvio 1. IoT-järjestelmän rakenne.

Ensin määritellään käyttöliittymän, verkkopalvelimen ja mikrokontrollerin vaatimukset. Mikrokontrolleri kerää anturitietoa, joka muunnetaan luettavaan muotoon ja lähetetään verkkopalvelimelle. Valittu mikrokontrollerikortti käyttää WiFi-verkkoa yhdistyäkseen internetiin,

jonka jälkeen se muodostaa Transport Layer Security (TLS) -salausprotokollalla salatun TCP/IP-yhteyden palvelimeen. Mikrokontrollerissa on paikallinen verkkopalvelin, josta tapahtuu laitteen GPIO-määrittely eli pinnien ohjelmointi signaalin vastaanottajaksi tai lähettäjäksi. Samassa palvelussa tehdään myös WiFi-verkkoon yhdistyminen, IoT-asetukset ja OTA-päivitykset. Over-the-Air (OTA) on sulautetun järjestelmän päivitys, joka lähetetään langattomasti laitteelle. Mikrokontrolleriin on liitetty lämpö- ja kosteusanturi. Valittu mikrokontrolleri kuuluu ESP32-tuoteperheeseen. Mikrokontrolleri on yhteydessä verkkopalvelimeen, joka toimii tiedonvälittäjänä, käsittelijänä ja tallentaa tietoa SQL-tietokantaan.

Verkkopalvelimella on järjestelmän oma verkkopalvelu. Verkkopalvelin vastaa käyttöliittymästä lähetettyihin palvelimen tarjoamiin API-rajapintaa käyttäviin kutsuihin. Käyttöliittymälle kirjaudutaan käyttäen Azure Active Directory-kirjautumispalvelua. Käyttöliittymältä pystytään API-kutsuja käyttäen hakemaan tietoa tietokannasta ja antamaan käskyjä mikrokontrollerille. Käskyjen lähetys mikrokontrollerille tapahtuu MQTT-viestiprotokollaa käyttäen ja myös mikrokontrolleri lähettää dataa käyttäen MQTT-protokollaa. Käyttöliittymässä näytetään lämpötila ja kosteusmittaustuloksia mikrokontrollerin mittauspaikasta. Käyttöliittymässä näytetään IO-ohjauslaitteista magneettisten kytkinten tilatietoja.

Konstruktiivisen tutkimuksen tavoitteena on ratkaista käytännön ongelma luomalla uusi konkreettinen konstruktio, joka pohjautuu teoreettiseen ja empiiriseen tietoon yhdistäen teorian ja käytännön (Ojasalo ym., 2009, s. 65–66). Opinnäytetyö on luonteeltaan konstruktiivinen tutkimus. Opinnäytetyössä tutkimusongelmana on sopivan IoT-arkkitehtuurin valinta prototyyppiin sekä se, kuinka pitää IoT-järjestelmä modulaarisena, kun eri osia arkkitehtuurista pitää pystyä käyttämään tarvittaessa pilvipalvelussa tai fyysisellä palvelimella. Näitä ehtoja täyttäen pitää lisäksi varmistaa toteutuksen tietoturvallisuus. Tutkimuskysymykset ovat seuraavat:

1. Miten toteuttaa sopiva järjestelmäarkkitehtuuri kokeiluun?
2. Miten huomioida modulaarisuus IoT-arkkitehtuurissa, jotta se on mahdollisimman yleiskäytännöllinen?
3. Miten voidaan parantaa tietoturvaa IoT-arkkitehtuurissa?

Opinnäytetyössä on rajoituksia IoT-arkkitehtuurille ja järjestelmälle pilvipalveluihin liittyen. Tarvittaessa on palvelinta ja web-käyttöliittymää pystyttävä ajamaan paikallisella palvelimella omana instanssina tai Docker-imagena, jos pilvipalveluita ei voida käyttää tai niiden käyttö on estetty.

1.2 Kehittämismenetelmä

Ojasalo ym. (2015, s. 65–66) mukaan konstruktiivinen tutkimus yrittää ratkaista ongelmaa luomalla uuden tuotoksen vanhasta. Tässä lähestymistavassa yhdistyvät empiirinen tieto eli käytännön oppiminen sekä kokemus teoreettiseen tietoon ja kokemukseen. Tutkimuksen tuloksena saadaan tuotos, josta saadaan uutta tietoa ja kokemusta. Uusi konstruktio testataan kohdeorganisaatiossa, ja sen perusteella voidaan tehdä päätelmät liittyen toimivuuteen ja käyttöönottoon.

Kuviossa 2 on konstruktiivisen tutkimuksen prosessi. Prosessi alkaa valitsemalla mielekäs ongelma ja sen määrittelyllä. Lukan (2001) mukaan tutkijan tulisi pohtia ongelmaa teorian ja käytännön kannalta, jolloin ongelman ratkaisulla on käytännöllistä merkitystä. Toisessa vaiheessa kerätään teoreettista ja käytännöllistä tietoa tutkimukseen ja kehittämiskohteeseen liittyen. Samalla pyritään hankimaan syvällinen tutkimusaiheen tuntemus teoreettisesti ja käytännöllisesti. Kolmannessa vaiheessa pyritään innovoimaan ratkaisumalli ja kehittämään ongelman ratkaisu. Tämä on kriittinen vaihe, koska jos ratkaisevaa konstruktiota ei pystytä tekemään, niin projektia ei ole syytä jatkaa. Konstruktion toimivuus todetaan oikeassa ympäristössä, jossa nähdään ratkaisun oikeellisuus ja käytännöllisyys. Lopussa tehdään toteutuksen teoriakytkentä ja tarkastellaan uutuusarvoa sekä soveltamisalueen laajuutta.

Kuvio 2. Konstruktiivisen tutkimuksen prosessi (Luka, 2001).

Työssä noudatetaan soveltavin osin edellä kuvattua prosessia. Konstruktiivisen tutkimusmenetelmän seuraavat osat toteutuvat kehitystyössä: ongelman etsiminen, käytännöllisen ja teoreettisen tiedon hankinta sekä ratkaisujen laatiminen. Työssä toteutetaan prototyyppi, joka pystytettiin käyttöön todelliseen ympäristöön. Tämän takia testaaminen, teoriakytkentä, soveltamisalueen ja uutuusarvon tarkastelu voidaan tehdä kokonaan.

2 JÄRJESTELMÄARKKITEHTUURI

2.1 Tarkoitus

Arkkitehtuurin ensisijainen tarkoitus on tukea järjestelmän elinkaarta (Martin, 2018, s. 136–137). Hyvä arkkitehtuuri tekee järjestelmästä selkeän, helposti kehitettävän, hyvin ylläpidettävän ja mahdollistaa onnistuneen käyttöönoton. Hyvän arkkitehtuurin tavoitteena on minimoida järjestelmän käyttöiän kustannukset ja maksimoida ohjelmistokehittäjän tuottavuus. Järjestelmän arkkitehtuurilla on kuitenkin hyvin vähän vaikutusta siihen, miten järjestelmä toimii. On monia järjestelmiä, joissa on huonot arkkitehtuurit, ja ne toimivat silti hienosti. Näissä ongelmat eivät piile järjestelmän toiminnassa. Ongelmat esiintyvät niiden käyttöönotossa, ylläpidossa ja kehittämisessä. Tämä ei tarkoita sitä, että arkkitehtuurilla ei olisi roolia laitteen oikean toiminnan tukemisessa järjestelmänä. Sen arkkitehtuuri varmasti tekee ja sen rooli on kriittinen.

Ohjelmistoarkkitehtuurin tulisi olla riittävän korkealla abstraktiotasolla (Ingeno, 2018, s. 10–11). Tällöin monet sidosryhmät, joilla on vähän tai ei lainkaan ymmärrystä ohjelmistojärjestelmistä silti ymmärtävät heille oleelliset asiat. Eri sidosryhmillä on erilaisia näkökulmia ja prioriteetteja sen suhteen, mitä he haluavat tietää järjestelmän toiminnasta. Yhteisen kielen tarjoaminen ja arkkitehtuurisuunnittelun ohjeistus parantaa heidän yhteisymmärrystänsä arkkitehtuurista. Tämä on erityisen hyödyllistä suurien ja monimutkaisten järjestelmien osalta, joita olisi muuten liian vaikea ymmärtää täysin. Arkkitehtuurin ymmärtäminen on tarpeellista erityisesti silloin, kun vaatimusmäärittely ja muut varhaiset päätökset tehdään ohjelmistojärjestelmälle. Ohjelmistoarkkitehtuuri helpottaa neuvotteluja sekä keskusteluja ohjelmistojärjestelmästä.

Ohjelmistoarkkitehtuuri edellyttää tärkeiden valintojen tekemistä ohjelmistojärjestelmän rakenteesta (Bass ym., 2012, s.3–7). Tämä edellyttää päätöksiä järjestelmän komponenttien valinnasta sekä kuinka ne ovat vuorovaikutuksessa toistensa kanssa. Komponentit ovat järjestelmän modulaarisia yksiköitä, jotka kapseloivat tiettyjä toimintoja. Ne voivat olla yksittäisiä moduuleja, luokkia tai suurempia ohjelmistoyksiköitä. Hyvin suunnitellut komponentit edistävät modulaarisuutta ja uudelleenkäytettävyyttä. Tämä sisältää näiden

komponenttien ryhmittelemisen suurempiin osajärjestelmiin ja noudattamalla tiettyä arkkitehtonista tyyliä ohjaamaan tätä järjestelyä.

2.2 Järjestelmäarkkitehtuurin periaatteet

Hyvän järjestelmäarkkitehtuurin tunnuspiirteitä on riippumattomuus ulkoisista kirjastoista, toimijoista, tietokannoista, käyttöliittymästä ja järjestelmän hyvä testattavuus (Martin, 2018, s. 148). Neljä keskeistä teemaa, jotka järjestelmäarkkitehtuurin tulisi toteuttaa ovat sille asetetut vaatimukset, ylläpidettävyys, kehitettävyys ja käyttöönotto. Hyvän ohjelmistoarkkitehtuurin pitää tukea käyttötapauksia ja järjestelmän käytettävyyttä. Tämä tarkoittaa, että järjestelmän arkkitehtuurin on tuettava järjestelmän tarkoitusta. Jos järjestelmä on esimerkiksi ostoskorisovellus, niin arkkitehtuurin on tuettava ostoskorin käyttötapauksia. Tämä onkin arkkitehdin ensimmäinen huolenaihe ja arkkitehtuurin ensimmäinen prioriteetti.

Tärkeää ymmärtää teknologian riippuvuuden katkaisemisen merkitys, ja hän suosittelee koodiriippuvuuksien eristämistä ylläpidettävyyden parantamiseksi, jotta nämä voidaan tunnistaa ja poistaa (Feathersin, 2004, s. 197–198). Tällä estetään järjestelmien sisäisiä tiukkoja kytkentöjä, mikä puolestaan edistää modulaarista ja löyhästi kytkettyä arkkitehtuuria. Koodi määritellään vanhaksi koodiksi, jos se ei sisällä testejä (mts. 17). Testaamaton koodi on luonnostaan hankala ylläpitää. Testaamattomaan koodiin voidaan lisätä testejä ja muokata vanhaa koodia uudelleen. Tämä mahdollistaa parannusten tekemisen ilman virheiden riskiä.

Suunnittelussa tulisi tehdä prototyyppejä ja testata suunnittelua, koska tällä pystytään aikaisessa vaiheessa toteamaan idean toimivuutta (Thomas ja Hunt, 1999, s. 86–88). Hyvin määritellyn ohjelmistoarkkitehtuurin merkitys suuressa ja monimutkaisessa järjestelmässä on kriittinen. Tavoitteena on hyvällä arkkitehtuurisuunnittelulla varmistaa, että jokainen tiimi voi työskennellä järjestelmän eri osien parissa itsenäisesti ja eristyksissä häiritsemättä muiden tiimien työtä. Tämän saavuttamiseksi järjestelmä on jaettava selkeästi määritellyiksi komponenteiksi, joista jokainen voidaan kehittää itsenäisesti. Nämä komponentit voidaan sitten jakaa erillisiin ryhmiin, jolloin jokainen tiimi voi työskennellä omalla järjestelmän osallaan ilman, että heidän tarvitsee huolehtia muiden tiimien tekemästä työstä (Martin, 2018, s. 149–150). Tämä lähestymistapa auttaa minimoimaan ristiriitojen ja viivästysten

riskiä, koska jokainen tiimi voi keskittyä omaan järjestelmän osaansa ilman, että heidän tarvitsee koordinoida muiden tiimien kanssa. Lähestymistapa helpottaa myös muutosten tekemistä järjestelmään, koska arkkitehtuuri mahdollistaa jokaisen tiimin itsenäisen toiminnan.

Koodin tulisi olla selkeätä, koska turha monimutkaisuus vaikeuttaa koodin ymmärtämistä (Thomas ja Hunt, 1999, s. 108–109). Tämän tarkoituksena on parempi koodin ylläpidettävyys ja virheidenkorjaus. Yksinkertainen suunnittelu ei ainoastaan helpota ymmärtämistä vaan myös edistää ohjelmiston ulkoasua ja kestävyyttä. Käytännössä tämä tarkoittaa selkeiden muuttujanimien käyttöä, koodin modulointia ja yksinkertaisten ratkaisujen suosimista monimutkaisten sijaan.

Tavoitteena on saavuttaa järjestelmän välitön käyttöönotto. Välitön käyttöönotto tarkoittaa, että järjestelmä on heti otettavissa käytettäväksi, kun se on rakennettu ja ilman laajaa manuaalista konfigurointia tai asennusta (Martin, 2018, s. 151). Tämän saavuttamiseksi järjestelmä on jäsennettävä oikein sekä eristettävä komponenteiksi. Tämä sisältää pääkomponentteja, jotka yhdistävät koko järjestelmän. Pääkomponentit varmistavat muiden komponenttien oikein käynnistyksen, integroitumisen ja valvonnan. Järjestelmä järjestää itsensä oikein tällä tavalla. Hyvä arkkitehtuuri auttaa myös minimoimaan manuaalisen määrityksen ja asennuksen. Tämä mahdollistaa järjestelmän nopean ja tehokkaan käyttöönoton.

Suunnitellun arkkitehtuurin pitää vastata hankkeen tavoitteita ja vaatimuksia, jotta se voidaan arvioida (Bass ym., 2012, s. 375). Arviointiprosessissa analysoidaan vaihtoehtoja ja varmistetaan, että valittu arkkitehtuuri täyttää odotetut tavoitteet. Arkkitehtuurin arviointi on askel, jossa otetaan huomioon arkkitehtuurin keskeinen rooli projektin onnistumisessa. Tärkeää on tasapainottaa arvioinnin kustannukset ja siitä saatu hyöty. Tärkeintä on varmistaa, että arvioinnista saatavat hyödyt ovat suuremmat kuin investoitu aika ja raha. Tällöin arviointi on kustannustehokas ja kertoo arkkitehtuurin sopivuudesta tarkoitukseen.

2.3 Ylhäältä alas -lähestymistapa järjestelmäarkkitehtuurin suunnittelussa

Ylhäältä alas -lähestymistapa järjestelmäarkkitehtuurin suunnittelussa on menetelmä hajottaa monimutkainen järjestelmä pienempiin osioihin (Ingeno, 2018, s. 117–118). Tällöin osiot ovat paremmin hallittavissa aloittamalla korkean tason näkymästä ja siirtymällä vähitellen kohti yksityiskohtia. Tätä lähestymistapaa käytetään usein ohjelmistoarkkitehtuurin suunnittelussa luomaan selkeä käsitys järjestelmän vaatimuksista ja rajoituksista ennen ratkaisun suunnittelua. Ylhäältä alas -lähestymistapa alkaa korkean tason näkemyksestä järjestelmästä, jossa on mukana tavoitteet ja rajoitukset. Arkkitehtuuri jakautuu yhä pienempiin osiin, mikä johtaa lopulta järjestelmän yksityiskohtaiseen suunnitteluun. Tämän lähestymistavan avulla arkkitehti ymmärtää järjestelmän yleiset vaatimukset ja rajoitukset ennen yksityiskohtien käsittelyä, mikä auttaa varmistamaan, että suunnittelu vastaa järjestelmän ja sen sidosryhmien tarpeita.

Ylhäältä alas -lähestymistapa tarjoaa selkeän ja yksinkertaisen rakenteen järjestelmän suunnittelulle, mikä helpottaa sen ymmärtämistä ja ylläpitoa (Ingeno, 2018, s. 118–119). Tämä lähestymistapa voi auttaa tunnistamaan mahdolliset riskit ja haasteet suunnitteluprosessin varhaisessa vaiheessa, jolloin arkkitehti voi tehdä muutoksia ja parannuksia ennen kuin niistä tulee suuria ongelmia. Ylhäältä alas -lähestymistapa on hyödyllinen työkalu ohjelmistoarkkitehdeille, jotka haluavat luoda tehokkaita ja skaalautuvia järjestelmäarkkitehtuurisuunnitelmia.

Brooksin (1995, s. 54–58) mukaan alkuperäisen järjestelmän onnistuminen kasvattaa yliluottamusta, mikä saa kehittäjät tekemään liian kunnianhimoisia ja monimutkaisia suunnittelupäätöksiä myöhempää järjestelmää varten. Tämä taipumus lisää usein monimutkaisuutta, projektien viivästyksiä ja ennakoimattomien haasteiden vaaraa, kun kehittäjät pyrkivät integroimaan joukon ominaisuuksia ja parannuksia, jotka oli jo suunniteltu, mutta joita ei ole otettu käyttöön ensimmäisessä järjestelmässä. Järjestelmää ei tulisi täten liiallisesti suunnitella etukäteen vaan kohtuudella.

Arkkitehtuurin kuvauksen tärkeimmät komponentit ovat sen mallit. Mallit edustavat järjestelmän tärkeitä puolia ja välittävät niitä sidosryhmille (Rozanski ja Woods, 2011, s. 172). Hyvin muotoiltu malli on välttämätön, jotta sidosryhmät ymmärtävät arkkitehtuurin. Arkkitehtuurin kuvaus koostuu useista näkymistä, ja jokainen näkymä sisältää malleja,

periaatteita, standardeja ja sanastoa. Malleja on kolmea päätyyppiä: kaksi on muodollista ja yksi epävirallinen. Muodollisiin malleihin kuuluvat laadulliset mallit, jotka kuvaavat keskeisiä rakenteellisia tai käyttäytymiselementtejä. Toiseksi määrällisiä malleja, jotka antavat lausuntoja järjestelmän mitattavissa olevista näkökohdista. Arkkitehdit keskittyvät usein laadullisiin malleihin, koska luotettavaa määrällistä analyysiä varten ei useinkaan ole yksityiskohtaista tietoa. Epävirallisia malleja ovat luonnokset. Luonnoksia käytetään pääasiassa viestintään vähemmän teknisten sidosryhmien kanssa.

2.4 Alhaalta ylös -lähestymistapa järjestelmäarkkitehtuurin suunnittelussa

Alhaalta ylöspäin suuntautuva arkkitehtuuri on ohjelmistoarkkitehtuurin suunnittelumalli, joka alkaa pienemmistä osista ja etenee kohti suurempia kokonaisuuksia. Tämä lähestymistapa korostaa yksityiskohtien ensisijaista suunnittelua ja implementointia ennen kuin siirrytään kohti järjestelmän laajempaa rakennetta. Tämä on samankaltainen kuin palapelissä pelaaminen, jossa pienet palat liitetään yhteen, jotta ne muodostavat suuremman kuvan.

Tärkein etu alhaalta ylöspäin suuntautuvassa arkkitehtuurissa on yksityiskohtien huolellinen hahmottaminen (Rozanski & Woods, 2011, s. 10–12). Tämä helpottaa pienten osien suunnittelua ja testaamista perusteellisesti, jolloin on mahdollista välttää ongelmia, jotka saattaisivat ilmetä myöhemmissä vaiheissa. Alhaalta ylöspäin suuntautuva suunnittelu tukee hyvin iteratiivista kehitystapaa. Tämän takia järjestelmää voidaan kehittää vaiheittain, jossa on joustavuutta muutosten tekemiseen. Tämä voidaan myös nähdä alhaalta ylöspäin suuntautuvana arkkitehtuurina. Tämä muistuttaa rakenna ja testaa -lähestymistapana. Tämä tarkoittaa, että suunnittelijat ja kehittäjät aloittavat projektin luomalla pieniä toiminnallisuuksia ja komponentteja, jotka täyttävät järjestelmän perusvaatimukset. Näitä pieniä palasia yhdistetään sitten vähitellen suuremmiksi kokonaisuuksiksi.

Tämä lähestymistapa antaa kehittäjille mahdollisuuden keskittyä yksittäisiin toiminnallisuuksiin ja niiden teknisiin yksityiskohtiin (Ingeno, 2018, s. 119–120). He voivat testata ja varmistaa, että kunkin osan toiminta on oikein ennen kuin he siirtyvät seuraavaan vaiheeseen. Kaikki pienet palaset voidaan tehdä valmiiksi, minkä jälkeen ne voidaan yhdistää

yhteen, jolloin ne muodostavat muodostaen kokonaisen ohjelmiston. Tämä mahdollistaa vähemmän riskialttiin ja paremmin testatun tuotteen kehittämisen.

Tarkentamalla ohjelmistoarkkitehtuuria jakamalla sitä komponentteihin alkaa järjestelmän rakenne selkiytyä, kun valitaan vaatimusmallissa hahmotettuja luokkia (Pressman, 2009, s. 258). Nämä analyysiluokat edustavat sovelluksen kokonaisuuksia, jotka tarvitsevat huomiota ohjelmistoarkkitehtuurissa. Yksi komponenttien johtamisen ja jalostamisen lähde on sovellusalue. Toinen lähde on infrastruktuuritoimialue. Arkkitehtuurissa on oltava erilaisia infrastruktuurikomponentteja, jotka tukevat sovelluskomponentteja, mutta eivät liity sovelluksen toimintaan. Esimerkiksi tällaisia komponentteja ovat muistinhallinta, viestintä, tietokanta ja tehtävien hallinta, jotka on usein integroitu ohjelmistoarkkitehtuuriin.

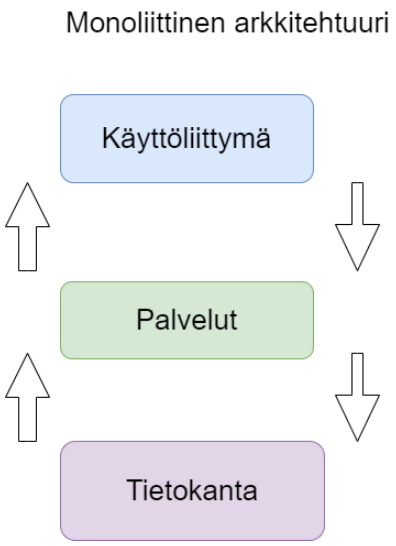
2.5 Monoliittinen järjestelmäarkkitehtuurimalli

Monoliittinen järjestelmäarkkitehtuuri viittaa perinteiseen ohjelmistoarkkitehtuuriin, jossa kaikki järjestelmän komponentit on tiiviisti kytketty ja integroitu yhdeksi, yhtenäiseksi suoritettavaksi kokonaisuudeksi. Tässä suunnittelussa koko sovellus on rakennettu ja otettu käyttöön yhtenä yksikkönä, jolloin järjestelmän muutokset edellyttävät koko sovelluksen uudelleenasennusta. Monoliittinen arkkitehtuuri oli normi varhaisissa ohjelmistojärjestelmissä ja sen yksinkertaisuus teki siitä suositun pienissä ja keskisuurissa projekteissa (Strimbei, 2015, s. 14). Kuitenkin järjestelmien kasvaessa ja muuttuessa monimutkaisemmiksi monoliittisen arkkitehtuurin rajoitukset tulivat ilmeisiksi. Järjestelmään tehdyt muutokset vaativat koko sovelluksen uudelleen rakentamista ja uudelleenasennusta, mikä vaikeutti uusien ominaisuuksien muokkaamista tai lisäämistä vaikuttamatta koko järjestelmään.

Lisäksi monoliittisia järjestelmiä on vaikea testata ja ylläpitää, koska yhden komponentin muutokset voivat vaikuttaa koko järjestelmään (Strimbei ym., 2015, s. 14). Tämä voi tehdä virheiden tunnistamisesta ja korjaamisesta haastavaa sekä voi myös vaikeuttaa järjestelmän skaalaamista vastaamaan lisääntynyttä käyttöä, vaikka monoliittista arkkitehtuuria voidaan edelleen käyttää yksinkertaisen käyttötarkoituksen sovelluksissa, pienissä kehitystiimeissä tai prototyyppivaiheessa. Monoliittinen arkkitehtuuri on suurelta osin korvattu modulaarisemmilla ja skaalautuvilla arkkitehtuurimalleilla, kuten mikropalveluilla ja

palvelukeskeisellä arkkitehtuurilla, jotka mahdollistavat joustavammat ja ylläpidettävät ohjelmistojärjestelmät.

Kuviossa 3 on esimerkki monoliittisesta arkkitehtuurista, jossa arkkitehtuuri on järjestetty kolmeen kerrokseen: tietovarastot alhaalla, järjestelmäpalvelut keskellä ja käyttöliittymäkerros ylhäällä. Tietovarastokerros sisältää järjestelmän tiedon tallennus- ja hallintakomponentit. Tietovarastokerros on vastuussa pysyvistä tiedoista ja sen saattamisesta muiden järjestelmän kerrosten saataville. Järjestelmäpalvelukerros, joka tunnetaan myös nimellä logiikkakerros. Logiikkakerros vastaa järjestelmän liiketoimintalogiikan toteuttamisesta. Järjestelmäpalvelukerros sisältää komponentteja, jotka käsittelevät tietoja, suorittavat laskelmia ja valvovat liiketoimintasääntöjä. Järjestelmäpalvelukerros on usein järjestelmän ydin ja on vuorovaikutuksessa tietovarastokerroksen kanssa, josta se hakee ja tallentaa tietoja. Käyttöliittymäkerros vastaa järjestelmän toimivuuden esittelemisestä käyttäjälle. Käyttöliittymäkerros sisältää komponentit, joiden avulla käyttäjät voivat olla vuorovaikutuksessa järjestelmän kanssa, kuten lomakkeet, painikkeet ja muut käyttöliittymäelementit.



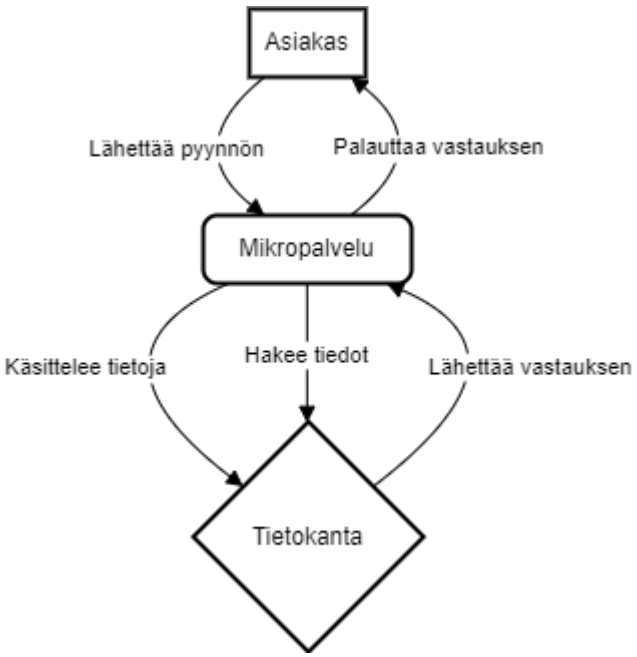
Kuvio 3. Esimerkki monoliittisesta arkkitehtuurista.

2.6 Mikropalvelut

Mikropalvelut ovat eräänlainen ohjelmistoarkkitehtuuri, joka keskittyy jakamaan sovelluksia pieniin itsenäisiin palveluihin (Aroraa ym., 2017, s. 7). Näiden palveluiden suunnittelu ja kehittäminen on tehty mahdollisimman tehokkaaksi, eikä niiden toimivuus ole riippuvainen

muista palveluista. Jokainen mikropalvelu on pieni itsenäinen palvelu, joka on kehitetty yhtä tarkoitusta varten. Tämä mahdollistaa enemmän joustavuutta ja skaalautuvuutta, koska jokaista palvelua voidaan muokata ja päivittää ilman, että se vaikuttaa koko sovellukseen. Lisäksi mikropalveluille on tunnusomaista se, että ne ovat pieniä liiketoimintalogiikaltaan, mutta eivät lähdekoodin pituuden tai komponenttien lukumäärän suhteen (Richardson, 2019, s. 19–20). Jakamalla sovellus erillisiin palveluihin niiden ylläpito helpottuu.

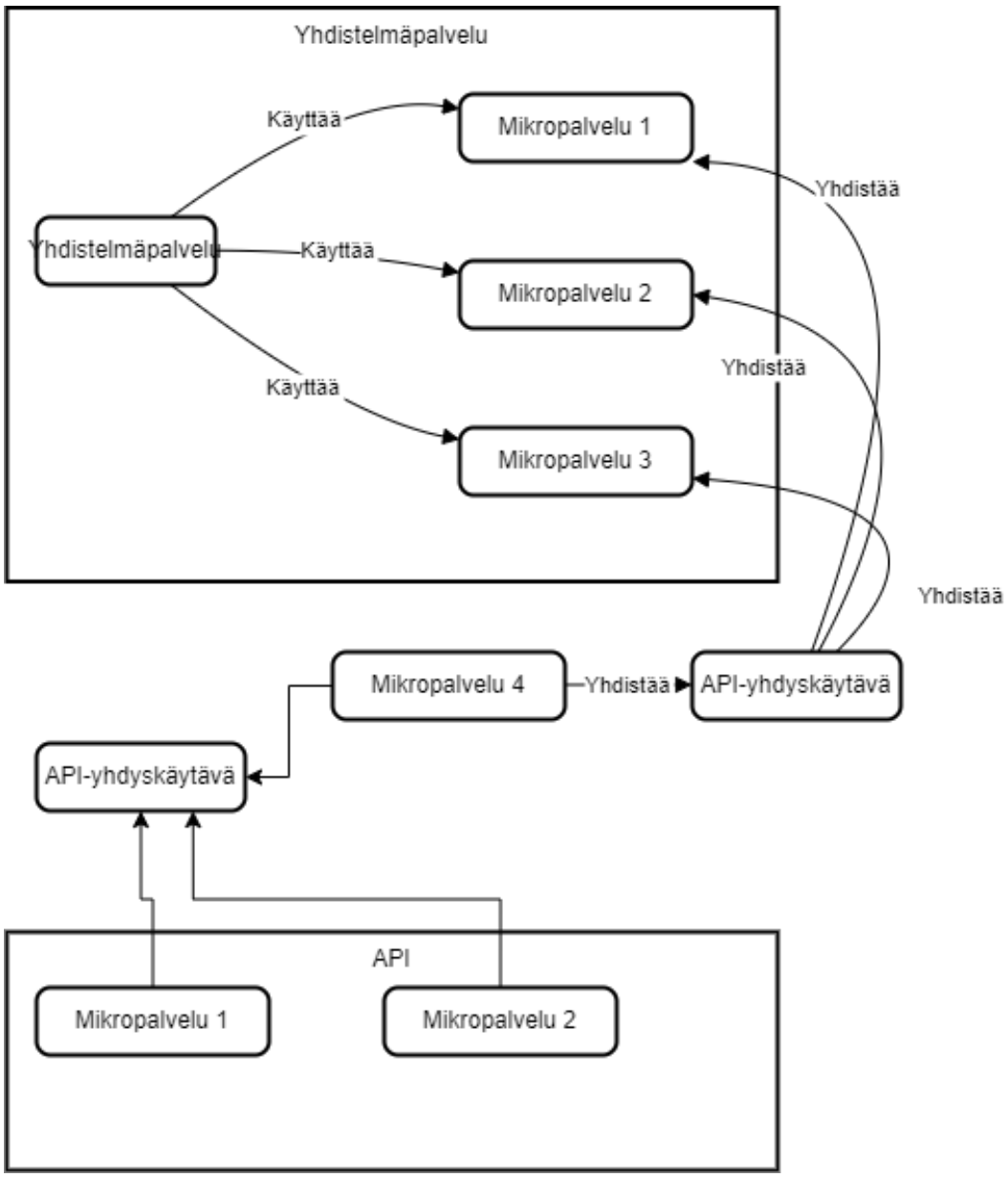
Kuviossa 4 on esimerkki mikropalvelusta, jossa asiakas lähettää mikropalvelulle pyynnön. Mikropalvelu välittää pyynnön eteenpäin tietokantaan. Tietokanta vastaa pyyntöön mikropalveluille, ja mikropalvelu lähettää sen vastauksena asiakkaalle.



Kuvio 4. Mikropalvelupohjaisen järjestelmäarkkitehtuurin rakenne.

Mikropalvelujen tärkein etu on, että ne mahdollistavat suuremman joustavuuden ja skaalautuvuuden. Jokaisen mikropalvelun päätehtävänä on täyttää yksittäinen osa liiketoiminnan vaatimuksia. Suurempia loogisia kokonaisuuksia voidaan rakentaa yhdistämällä yksi tai useampi mikropalvelu laajemmaksi prosessiksi (Subramanian & Raj, 2019, s. 137). Tämä voidaan tehdä kahdella eri tavalla, joko API-yhdyskäytävän tai yhdistelmäpalvelun kautta. API-yhdyskäytävän tapauksessa yhdyskäytävä yhdistää tarvittavan määrän mikropalveluita yhdeksi kutsuksi asiakasjärjestelmästä. Yhdistelmäpalvelu puolestaan hoitaa

tarvittavat mikropalvelukutsut. Molempien lähestymistapojen tavoitteena on varmistaa, että suurempi prosessi toteutetaan yhtenä kokonaisuutena asiakkaan näkökulmasta. Tällöin jokainen mikropalvelu säilyy itsenäisesti käyttöönotettavana ja skaalautuvana. Tällainen rakenne on esitetty kuviossa 5.



Kuvio 5. Erilaisten mikropalveluiden yhdistäminen.

Itsenäisyyden käsite on mikropalveluarkkitehtuurin kriittinen osa. Jokainen mikropalvelu on suunniteltu riippumattomaksi muista järjestelmän mikropalveluista, joten se voidaan asentaa ja päivittää vaikuttamatta muihin mikropalveluihin (Newman, 2015, s. 23). Löyhästi

kytkettyjen viestintäprotokollien, kuten RESTful APIn ja asynkronisen viestinnän käyttö mahdollistaa jokaisen mikropalvelun vuorovaikutuksen muiden palvelujen kanssa. Tällöin kommunikaatio voidaan toteuttaa tietokenttien kautta. Tämä tekee järjestelmästä vähemmän jäykän ja joustavamman muutoksille. Tämän riippumattomuuden ansiosta jokainen mikropalvelu voidaan ottaa käyttöön erillisenä palveluna ja integroida suurempaan järjestelmään tai tarjota pilvialustan palveluna. Tämä on olennaista luotaessa modulaarista ja skaalautuvaa järjestelmää, joka mukautuu muuttuviin vaatimuksiin ja mahdollistaa yksittäisten palvelujen helpomman ylläpidon sekä päivittämisen vaikuttamatta muuhun järjestelmään.

Monoliittisessa järjestelmässä teknologian muutos edellyttää, että koko järjestelmä tai suuri osa siitä muutetaan uuden teknologian hyödyntämiseksi (Newman, 2015, s. 24). Tämä voi olla hidas, aikaa vievä ja kallis prosessi, jolla on huomattavia kustannuksia. Lisäksi uuden teknologian toimivuuden varmistaminen edellyttää muutosten tekemistä järjestelmän moniin eri osiin. Mikropalveluiden avulla käytettävän teknologian tehtyä valintaa voidaan kuitenkin muuttaa helpommin. Mikropalvelut kertovat soveltuuko valittu tekniikka täyttämään asetetut vaatimukset ja onko siitä hyötyä. Monoliittisessa järjestelmässä koko järjestelmään lisätään tyypillisesti lisäresursseja, kun se joutuu raskaan rasituksen alle. Tämä voi johtaa yli- tai aliresursointiin. Mikropalveluilla resurssit voidaan puolestaan skaalata erikseen kullekin palvelulle, mikä optimoi resurssien käytön ja alentaa kustannuksia. Tämä varmistaa, että vain tarvittavat resurssit lisätään tiettyihin järjestelmän osiin.

Mikropalveluarkkitehtuuri tuo etuja ja tehokkuutta järjestelmän ylläpidon kannalta. Mikropalveluarkkitehtuurissa jokainen mikropalvelu voidaan päivittää ja asentaa erikseen, mikä vähentää koko järjestelmän seisokkiaikaa ja nopeuttaa päivitysprosessia (Newman, 2015, s. 27). Vikatilanteen sattuessa se on myös helpompi käsitellä, koska ongelma voidaan eristää tietylle mikropalvelulle ja korjaavat toimenpiteet voidaan kohdistaa vain ongelmapalveluun. Tämä nopeuttaa ja tehostaa ongelmien tunnistamis- ja korjausprosessia sekä vähentää riskiä, että yksittäinen vika vaikuttaa koko järjestelmän.

2.7 Tietoturva arkkitehtuurin osana

Tietoturvalla ohjelmistoarkkitehtuurissa tarkoitetaan käytäntöä suunnitella ja toteuttaa tietoturvatoimenpiteitä arkkitehtuurin tietojen ja järjestelmien luottamuksellisuuden, eheyden ja saatavuuden suojaamiseksi (Brown, 2014, s. 7). Tämä sisältää mahdollisten tietoturvariskien tunnistamisen ja niiden vaikutusten arvioinnin sekä asianmukaisten turvatoimien toteuttamisen riskien estämiseksi tai lieventämiseksi. Tehokas tietoturva ohjelmistoarkkitehtuurissa edellyttää perusteellista ymmärrystä järjestelmän suunnittelusta ja toteutuksesta sekä kattavaa tietämystä ajankohtaisista tietoturvauhkista ja haavoittuvuuksista. Huomioitavia ovat myös järjestelmään liittyvät eri sidosryhmät. Näitä ovat käyttäjät, järjestelmänvalvojat ja kolmannen osapuolen palveluntarjoajat.

Tietoturva on ohjelmistoarkkitehtuurin keskeinen osa-alue, koska se voi suoraan vaikuttaa järjestelmän yleiseen luotettavuuteen (Brown, 2014, s. 188–190). Turvallisuusuhat kehittyvät jatkuvasti ja siksi arkkitehtien on pysyttävä ajan tasalla uusimpien uhkien ja parhaiden käytäntöjen kanssa. Tämä mahdollistaa myös tehokkaiden turvatoimien suunnittelun. Turvallisuusnäkökohdat tulisi sisällyttää kaikkiin arkkitehtuurin osa-alueisiin aina alkuperäisestä suunnittelusta jatkuvaan ylläpitoon ja päivityksiin. Tähän tulisi sisällyttää säännölliset tietoturva-arvioinnit sekä mahdollisten tietoturvaloukkausten tarkkailu. Samalla tulisi säännöllisesti tarkistaa, että suojaustoiminnot on määritetty oikein ja ne ovat ajan tasalla.

Tietoturva on olennainen osa ohjelmistoarkkitehtuuria, sillä se suojaa arkaluontoisia ja arvokkaita tietoja kyberuhkilta, jotka voivat johtaa merkittäviin taloudellisiin menetyksiin ja vaikuttaa maineeseen (Bass ym., 2012, s. 384–386). Ensisijainen tavoite ohjelmistoarkkitehtuurissa on varmistaa ohjelmiston ja tietojen luottamuksellisuus, eheys ja saatavuus. Luottamuksellisuus varmistaa, että vain valtuutetuilla käyttäjillä on pääsy arkaluonteisiin tietoihin. Tietojen eheys takaa, että tiedot ovat tarkkoja ja niitä ei ole käsitelty. Saatavuus varmistaa, että tiedot ovat tarvittaessa valtuutettujen käyttäjien saatavilla. Suositeltavaa on turvallisuuden etusijalle asettava suunnittelu, jossa tietoturvariskit tunnistetaan ja niihin puututaan varhaisessa kehitysprosessissa sen sijaan, että niihin puututtaisiin vain tietoturvahäiriöiden sattuessa.

Toinen tärkeä tietoturvanäkökohta ohjelmistoarkkitehtuurissa on riskienhallinta. Tämä sisältää ohjelmistoon ja tietoihin liittyvien riskien tunnistamisen ja arvioinnin sekä

asianmukaisten suojaustoimien toteuttamisen kyseisten riskien vähentämiseksi. Suojaustoiminnot voivat sisältää kulunvalvontaa, salausta, palomuuria, tunkeutumisen havaitsemis- ja estojärjestelmiä sekä säännöllisiä varmuuskopioita (Bass ym., 2012, s. 387–388).

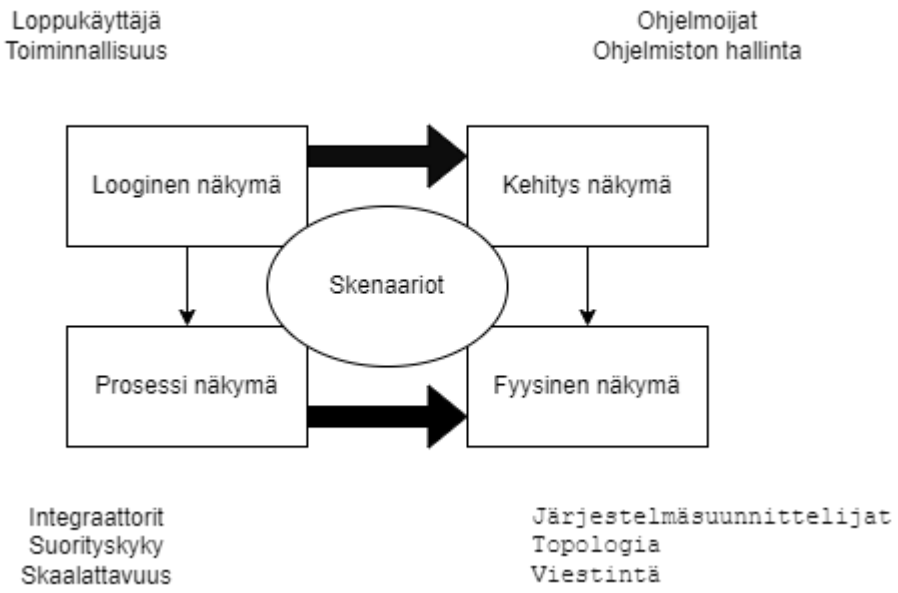
2.8 Dokumentointi

Järjestelmäarkkitehtuurin dokumentaation tarkoitus on kuvata järjestelmän toimivuutta ja toimia ohjenuorana järjestelmän kehittämiselle (Ingeno, 2018, s. 138). Dokumentaation tulee vastata järjestelmän suunnitteluvaiheessa tehtyjä päätöksiä. Arkkitehtuurinäkymät ovat ohjelmistojärjestelmän arkkitehtuurin erikoistuneita esityksiä, joista jokainen keskittyy tiettyihin huolenaiheisiin tai näkökohtiin, kuten toimivuuteen, rakenteeseen tai käyttöön. Ne auttavat sidosryhmiä ymmärtämään, viestimään ja dokumentoimaan arkkitehtuuria paremmin tarjoamalla näkökulmia heidän tarpeisiinsa. Järjestelmäarkkitehtuuria kuvataan tyypillisesti käyttämällä arkkitehtuurinäkymiä, koska yleinen arkkitehtuuri on usein monimutkainen. Sen takia se sisältää monia erilaisia komponentteja, jotka eivät mahdu järkevästi yhteen kuvaan. Nämä näkymät tarjoavat tavan kuvata järjestelmäarkkitehtuuria ja sen eri komponentteja selkeästi ja kattavasti.

Arkkitehtuuristen näkymien ja kaavioiden luontiprosessi alkaa suunnittelemalla järjestelmäarkkitehtuurin yleiset ääriviivat. Nämä ääriviivat toimivat ikään kuin suunnittelun raameina ja niiden tulee olla selkeitä sekä palvella tiettyä tarkoitusta. Tärkeää on ymmärtää, että nämä ääriviivat eivät sido tiettyyn ennalta määrättyyn muotoon, vaan niitä voidaan soveltaa tarpeiden mukaan. Kun yleiset linjaukset ovat asetettu, seuraava vaihe on tarkempi suunnittelu. Tässä vaiheessa keskitytään yksityiskohtiin, kuten järjestelmän osiin ja niiden välisten suhteiden kuvaamiseen. Tärkeää on dokumentoida tehdyt päätökset ja niiden perustelut.

Ohjelmistoarkkitehtuurin 4+1-näkymämalli esittelee kattavan lähestymistavan ohjelmistoarkkitehtuuriin 4+1-näkymämallin kautta (Kruchten, 1995, s. 1–2). Tässä mallissa on viisi samanaikaista näkymää, joita ovat looginen näkymä, prosessinäkymä, kehitysnäkymä ja skenaariot tai käyttötapaukset. Looginen näkymä kuvaa järjestelmän toimintoja loppukäyttäjille. Prosessinäkymä havainnollistaa dynaamisia tapahtumia keskittyen samanaikaisuuteen ja synkronointiin. Kehitysnäkymässä korostetaan ohjelmiston organisointia

moduuleiksi. Skenaariot tai käyttötapaukset tarjoavat käyttökertomuksia. Kuvio 6 esittelee kyseisen mallin.



Kuvio 6. Ohjelmistoarkkitehtuurin 4+1-näkymämalli (soveltaen Kruchten,1995).

Näkymät ovat välttämättömiä arkkitehtuuristen ongelmien monimutkaisuuden hallinnassa, mutta he korostavat mahdollisia sudenkuoppia (Rozanski & Woods, 2011, s. 34). Yksi näistä on epäjohdonmukaisuudet, koska arkkitehtuurisen poikkinäkymän johdonmukaisuuden varmistaminen on välttämätöntä. Sopivan näkymätavan valintaa pidetään tärkeänä tehtävänä, johon vaikuttaa arkkitehdin kokemus, sidosryhmien kokemus ja aikarajoitukset. Toisena haasteena on pirstoutumisen riski, jossa liiallinen määrä toisistaan riippumattomia näkymiä tekee arkkitehtuurista vaikeasti seurattavan. Tämän vähentämiseksi poistetaan tarpeettomia näkymiä, ja joissakin tapauksissa hybridinäkymien luomista useiden ongelmien ratkaisemiseksi. Tässä suositellaan varovaisuutta, jotta nämä yhdistetyt näkemykset eivät muutu liian monimutkaiseksi.

Päätösten dokumentointi on tärkeää, sillä se varmistaa, että muut tiimin jäsenet ja ulkopuoliset voivat ymmärtää, miksi tietyt ratkaisut on valittu ja mitä tarkoituksia ne palvelevat. Näin suunnitteluprosessi pysyy läpinäkyvänä ja päätöksentekoprosessiin voi palata tarvittaessa myöhemmin (Ingeno, 2018, s. 139).

Jayaprakashin (2008, s. 294–296) mukaan hyvin laadittu dokumentaatio on:

- **Selkeä:** Hyvä dokumentointi tarjoaa selkeän ja helposti ymmärrettävän kuvauksen rakenteesta sekä toiminnasta. Tämä selkeys auttaa ymmärtämään dokumentoitua kokonaisuutta ja sen komponentteja.

- **Viestii tehokkaasti:** Laadukas dokumentaatio helpottaa tiedon jakamista. Tämä luo yhteisen kielen, joka edistää tehokasta viestintää.

- **Opas:** Hyvin dokumentoitu kokonaisuus toimii oppaana uusille ryhmän jäsenille. He voivat nopeasti perehtyä kokonaisuuden rakenteeseen ja toimintaan. Tämä taas säästää aikaa ja vaivaa.

- **Ongelman tunnistus:** Dokumentaatio auttaa tunnistamaan mahdolliset ongelmat ja virheet järjestelmässä. Jos muutoksia tarvitaan, silloin dokumentaatio helpottaa ennakoimaan niiden vaikutukset ja varmistamaan, ettei uusia ongelmia synny.

- **Uudelleenkäytettävä:** Laadukas dokumentaatio mahdollistaa aiemmin toimiviksi todettujen ratkaisujen uudelleenkäytön. Tämä parantaa tehokkuutta ja nopeuttaa uusien järjestelmien kehitystä.

- **Viestii tehokkaasti:** Hyvä dokumentaatio tukee tehokasta viestintää järjestelmän keskeisistä osista, päätöksistä ja suunnitelmista. Tämä on kriittistä projektin onnistumisen kannalta.

3 IoT – ESINEIDEN INTERNET

3.1 Yleisesti esineiden internet

Internet of Things (IoT) on verkoissa olevia laitteita, jotka mahdollistavat tietojen liikkumisen fyysisten objektien välillä (Buyya ym., 2016, s. 3–4). Tämä tarkoittaa käytännössä, että lähes minkä tahansa esineen tai laitteen voi liittää IoT-verkkoon. Tämä voi sisältää kaikkea kodinkoneista ja terveydenhuollon laitteista teollisuuslaitteisiin ja ympäristöantureihin. Liittämällä nämä fyysiset esineet verkkoon voidaan niillä kerätä tietoa niiden ympäristöstä. Nämä tiedot voidaan lähettää muihin laitteisiin tai pilvipalveluihin analysointia ja päätöksentekoa varten. Tämä mahdollistaa älykkäiden järjestelmien kehittämisen ja antaa meille mahdollisuuden hyödyntää tietoa monilla eri tavoilla, kuten hälyttiminä tai kulunvalvontana. IoT:n ydinajatuksena on luoda saumaton verkko, jossa nämä esineet kommunikoivat keskenään ja joskus ihmisten kanssa, jolloin he voivat aistia ympäristönsä, kerätä dataa ja ryhtyä toimiin sen perusteella.

IoT:n merkitys on sen kyvyssä parantaa teollisuusautomaatiota, älykkäitä kaupunkeja, terveydenhuoltoa, maataloutta ja ympäristön seurantaa (Buyya ym., 2016, s. 4–5). IoT-laitteiden tuottamaa dataa hyödyntämällä voidaan tehdä tietoisia päätöksiä ja optimoida prosesseja, mikä parantaa resurssien hallintaa, tuottavuutta ja yleistä hyötyä. IoT-teknologia kehittyy jatkuvasti ja tarjoaa monipuolisia sovelluksia ja etuja eri toimialoilla, mikä tekee siitä digitaalisen aikakauden kulmakiven.

3.2 Langaton tiedonsiirto WiFillä

WiFi on langaton tekniikka, jolloin laitteet voivat muodostaa yhteyden internetiin ilman fyysisiä kaapeleita (Miller, 2015, s. 85–86). Tämä tekee siitä kätevän vaihtoehdon IoT-laitteille, joita voidaan joutua siirtämään tai sijoittamaan eri paikkoihin. Tästä voi olla hyötyä IoT-sovelluksissa, jotka vaativat nopeaa tiedonsiirtoa, kuten etävalvontaa tai reaaliaikaista ohjausta. Monet laitteet on jo suunniteltu toimimaan WiFin kanssa, joten niiden integrointi IoT-järjestelmään voi olla suhteellisen helppoa.

WiFin käyttö IoT:ssä mahdollistaa laitteiden kommunikoinnin keskenään ja internetin kanssa reaaliajassa. Tämä mahdollistaa laajojen tietomäärien keräämisen ja analysoinnin, jonka avulla voidaan parantaa tehokkuutta, alentaa kustannuksia ja parantaa käyttökokemusta (Bahga & Madisetti, 2014, s. 110–112). Yksi WiFin käytön tärkeimmistä eduista IoT:ssä on sen nopeus ja luotettavuus. WiFi voi siirtää dataa suurilla nopeuksilla, mikä mahdollistaa reaaliaikaisen viestinnän laitteiden välillä. Se on myös laajalti saatavilla, joten se on helppokäyttöinen ja edullinen vaihtoehto monille IoT-sovelluksille. WiFiä voi kuitenkin rajoittaa myös sen kantama ja yhteen verkkoon kytkettävien laitteiden määrä.

WiFin käytön negatiivisia puolia IoT:ssa on kantaman rajoitukset, turvallisuusongelmat, häiriöt ja virrankulutuksen (Miller, 2015, s. 86–87). Seinät, häiriöt ja etäisyys voivat vaikuttaa WiFi-signaaleihin. Tämä voi rajoittaa WiFi-verkon kantamaa, mikä voi olla ongelma IoT-sovelluksissa, jotka vaativat laajan peittoalueen. WiFi-verkot voivat olla haavoittuvia hakkeroinnille ja muille tietoturvauhkille, jotka voivat vaarantaa IoT-järjestelmän tiedot ja toiminnallisuuden. Muiden laitteiden ja verkkojen häiriöt voivat vaikuttaa WiFi-verkkoihin. Tämä voi johtaa signaalin voimakkuuden heikkenemiseen tai tiedonsiirtovirheisiin. WiFi voi kuluttaa enemmän virtaa kuin muut langattomat tekniikat, mikä voi olla ongelma akkukäyttöisille IoT-laitteille.

3.3 TCP/IP-tiedonsiirtoprotokolla

Yksi TCP/IP-tiedonsiirtoprotokollan hyvistä puolista on sen yhteensopivuus ja joustavuus (Blank, 2004, s. 5–7). Sitä ei ole sidottu tiettyyn käyttöjärjestelmään tai laitteeseen, joten sitä voidaan käyttää lähes kaikissa laitteissa. TCP/IP on suunniteltu erittäin joustavaksi, jolloin IP-osoitteet voidaan määrittää uudelleen eri laitteille. Tämä tekee siitä suositun tiedonsiirtoprotokollan. Toinen tärkeä ominaisuus on tiedonsiirto eri laitteiden välillä. Tietoa tulisi voida siirtää useiden laitteiden välillä ilman ongelmia. Tietoa lähetettäessä tiedonsiirtopaketti kulkee useiden eri laitteiden, kuten verkkokytkimien, palomuurien ja reittien kautta. TCP/IP:llä ei ole rajoitusta reitityksen määrälle, joten se soveltuu erityisen hyvin tällaiseen viestintään. TCP/IP:n käytössä on myös haittoja. Esimerkiksi se voi olla haavoittuvainen turvallisuusuhkille ja saattaa edellyttää lisäturvatoimenpiteiden käyttöönottoa. Lisäksi TCP/IP:n joustavuus voi joskus vaikeuttaa esiin tulevien ongelmien vianmääritystä.

TCP-yhteyden muodostaminen vaatii kolmisuuntaisen kättelyn, missä teknisiä tietoja vaihdetaan ja vasta tämän jälkeen tiedonsiirto voi alkaa. Tällä taataan luotettavan tiedonsiirto, missä katoavat datapaketit lähetetään uudelleen varmistaen, että tiedot toimitetaan varmasti (Blank, 2004, s. 15).

- **Kolmisuuntainen kättely:** Ennen kuin varsinaisia tietoja voidaan lähettää, lähettäjä ja vastaanottaja osallistuvat kolmisuuntaiseen kättelyprosessiin. Tämä vaihe on välttämätön yhteyden muodostamiseksi ja sisältää vaiheen, missä teknisiä tietoja vaihdetaan yhteyden parametrien määrittämiseksi. Tietojen lähettämistä ei vielä tapahdu tässä vaiheessa (Blank, 2004, s. 16).

- **Tiedonsiirto:** Kun kolmisuuntainen kättely on onnistuneesti suoritettu ja yhteys on muodostettu, tiedonsiirto voi alkaa. Lähettäjä alkaa lähettää varsinaisia tietoja ja vastaanottaja vastaanottaa sekä prosessoi näitä tietoja (Blank, 2004, s. 16).

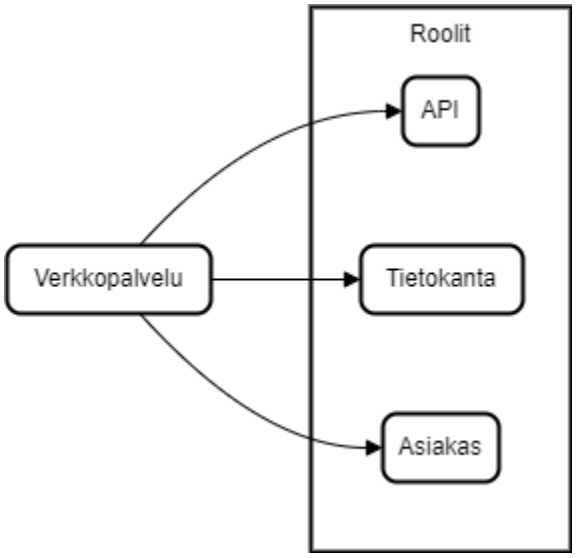
- **Datasegmenttien kuittaus:** Vastaanottaja lähettää takaisin kuittauksen (ACK) jokaisen vastaanotetun datasegmentin kohdalla. Tämä kuittaaminen ilmoittaa lähettäjälle, että kyseinen datasegmentti on turvallisesti vastaanotettu (Blank, 2004, s. 16).

- **Häviämisen korjaus:** Jos jokin datasegmentti katoaa tai tulee vaurioituneena tiedonsiirron aikana, niin tällöin sekvenssinumerointi ja kuittaamismekanismi auttavat havaitsemaan puuttuvat tai vahingoittuneet segmentit. Silloin puuttuvat datasegmentit lähetetään uudelleen, että kaikki tiedot toimitetaan oikein ja täydellisinä (Blank, 2004, s. 16).

- **Yhteyden katkaisu:** Tiedonsiirto katkaistaan, kun yhteys on valmis. Tämä tapahtuu nelisuuntaisella kättelyllä, joka varmistaa, että sekä lähettäjällä että vastaanottajalla ei enää ole lisää dataa lähetettäväksi ja yhteys suljetaan turvallisesti (Blank, 2004, s. 16).

3.4 Verkkopalvelu

Verkkopalvelu sisältää kolme keskeistä roolia: palvelun julkaisija, palvelun kuluttaja ja palvelurekisteri (Balani & Hathi, 2009, s. 35). Palvelun julkaisija vastaa palvelukuvauksien julkaisemisesta palvelurekisteriin ja mahdollistaa näin palvelujen käytettävyyden palvelun kuluttajille. Palvelujen kuluttajat voivat tarkistaa palvelut, jotka ovat saatavilla palvelurekisterissä ja kutsua näitä palveluja niiden kuvauksien perusteella. Tiedonvaihdossa yksinkertaisten viestien lähettämiseen ja vastaanottamiseen REST-arkkitehtuurityyli on yleisesti käytetty. Kun vaaditaan monimutkaisempaa tiedonvaihtoa useiden järjestelmien välillä tai viestin turvallisuutta, silloin SOAP-tyyli on yleinen valinta. Verkkopalvelut on suunniteltu helpottamaan eri järjestelmien ja laitteiden välistä kommunikaatiota määrittelemällä yhteiset säännöt ja standardit tiedonvaihdolle. Lopullinen teknologian valinta riippuu järjestelmävaatimuksista. Esimerkki verkkopalvelusta on kuviossa 7.



Kuvio 7. Esimerkki verkkopalvelusta.

3.4.1 SOAP & REST verkkopalveluissa

SOAP (Simple Object Access Protocol) on verkkopalvelun tiedon kommunikointiprotokolla, joka perustuu XML:ään (Jorgensen, 2002, s. 136–139). XML (Extensible Markup Language) on monipuolinen ja alustasta riippumaton merkintäkieli, jota käytetään määrittämään ja jäsentämään tietoja tallennusta, vaihtoa ja esittämistä varten. Se tarjoaa järjestelmille standardoidun tavan vaihtaa tietoja keskenään riippumatta kunkin järjestelmän

käyttämästä ohjelmointikielestä tai käyttöjärjestelmästä. SOAP ei ole sidottu tiettyyn tiedonsiirtoteknologiaan, mutta yleisin tapa siirtää SOAP-viestejä on HTTP-yhteys (Balani & Hathi, 2009, s. 35). Palveluntarjoaja julkaisee käytettävät palvelut WSDL (Web Service Description Language) -kuvaustiedoston kautta. WSDL-tiedosto sisältää teknisen kuvauksen siitä, kuinka järjestelmän eri palveluita voidaan kutsua ja missä osoitteessa niitä tarjotaan. Palveluntarjoaja julkaisee käytettävät palvelut WSDL (Web Service Description Language) -kielellä kuvattuna palvelurekisteriin. SOAP tarjoaa standardoidun tavan lähettää viestejä järjestelmien välillä sekä sitä käytetään yritystason sovelluksissa järjestelmien integrointiin. Se voi kuitenkin olla monimutkaisempi kuin muut verkkopalveluprotokollat ja vaatii enemmän käsittelyä sekä kaistanleveyttä.

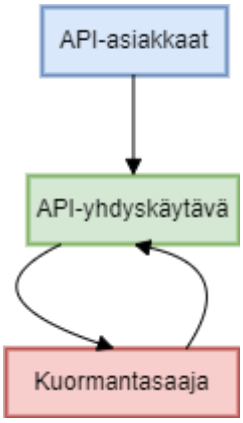
REST perustuu ajatukseen resursseista, jotka tunnistetaan URIlla (Kanjilal ja Joydip, 2013, s. 8). Resurssit voivat edustaa esimerkiksi asiakasta, tilausta tai tuotetta. Jokaisella resurssilla on tila, joka voidaan esittää esimerkiksi XML- tai JSON-muodossa. JSON on avoimen standardin muotoinen tiedonsiirtomuoto, jota ihmisten on helppo lukea ja kirjoittaa sekä koneiden jäsentää ja luoda. REST-viestintä asiakkaan ja palvelimen välillä tapahtuu HTTP-protokollan kautta. Asiakas lähettää palvelimelle pyyntöjä, jotka koostuvat HTTP-kutsusta ja URIsta, joka tunnistaa resurssin. Palvelin vastaa HTTP-tilakoodilla ja resurssin esityksellä pyydetyssä muodossa (Subramanian & Raj, 2019, s. 17–18). RESTtissä jokainen asiakkaan pyyntö sisältää kaikki pyynnön suorittamiseen tarvittavat tiedot, eikä palvelin säilytä asiakastilaa pyyntöjen välillä. Tämä tekee REST-palveluista helposti skaalattavissa ja mahdollistaa paremman suorituskyvyn sekä luotettavuuden. RESTtiä käytetään laajalti verkkosovelluksissa ja mobiilisovelluksissa sekä sitä käytetään myös muilla aloilla, kuten IoT ja pilvipalveluissa.

3.4.2 Verkkopalvelun valvonta ja vikasietoisuus

API-yhdyskäytävän valvonta ja vikasietoisuus ovat osa järjestelmää. API-yhdyskäytävä toimii kaiken viestinnän keskuksena ja sen häiriöt voivat vaikuttaa koko järjestelmään. Siksi on tärkeää seurata ja valvoa API-yhdyskäytävää, että järjestelmä toimii luotettavasti. API-yhdyskäytävän seuranta tarjoaa arvokasta tietoa järjestelmän suorituskyvystä ja toimivuudesta. Tämä valvonta voi auttaa ennakoimaan mahdollisia käyttöasteen nousuja ja varmistamaan, että resurssit ovat riittävät (Subramanian & Raj, 2019, s. 150).

Yksittäinen vika API-yhdyskäytävässä voi aiheuttaa tilanteen, jossa koko järjestelmä voi pysähtyä tai tulla käyttökelvottomaksi toimintahäiriön vuoksi (Subramanian & Raj, 2019, s. 138). Tahattomat häiriöt voivat aiheuttaa järjestelmän kaatumiseen. Tahattomia häiriöitä ovat esimerkiksi riittämättömät resurssit, jotka aiheuttavat palvelun ylikuormituksen. Tahallisia häiriöitä ovat esimerkiksi haitalliset hyökkäykset tai hakkerointiyritykset. Tahalliset häiriöt voivat myös häiritä järjestelmää taloudellisista tai muista syistä.

Kuormantasaaja vastaa eri API-yhdyskäytävien valvonnasta ja jakaa viestiliikenteen aktiivisten yhdyskäytävien kautta ennalta määritettyjen sääntöjen perusteella (Subramanian & Raj, 2019, s. 138). Kuormantasaaja suorittaa myös omaa valvontaa API-yhdyskäytäville havaitakseen vikoja tai ongelmia, eikä lähetä viestejä API-yhdyskäytävään, joka on vikatilassa. Kun API-yhdyskäytävä on korjattu ja toimii kunnolla, alkaa kuormituksen tasapainotin lähettämään sille viestejä uudelleen. Kuviossa 8 on esimerkki tästä.



Kuvio 8. API-kuorman jako verkkopalvelussa.

3.5 MQTT-viestintä

MQTT (Message Queuing Telemetry Transport) on viestintäprotokolla, jonka IBM on alun perin suunnitellut avoimeksi, yksinkertaiseksi, kevyeksi ja helposti kehitettäviksi. Se on asynkroninen julkaisija ja tilaajaprotokolla, joka toimii TCP-pinon päällä (Karagiannis ym., 2015, s. 4). MQTT:n avulla laitteet voivat vaihtaa viestejä keskenään ilman jatkuvaa yhteyttä, mikä on tärkeää resurssirajoitteisille IoT-laitteille, joilla voi olla rajoitettu akun käyttöikä tai verkkoyhteys. MQTT:n julkaisija ja tilaaja toiminnallisuus sopii erityisen hyvin IoT-laitteille, koska se mahdollistaa viestien lähettämisen suoraan vastaanottajalaitteeseen

sen sijaan, että vastaanottajan sovellus vaadittaisiin hakemaan viesti. Tämä tunnetaan push-toiminnallisuutena ja se voi lisätä IoT-sovellusten vuorovaikutteisuutta. Samalla kyseinen ominaisuus vähentää resurssien käyttöä, koska tietoja ei tarvitse tarkistaa tietyin väliajoin.

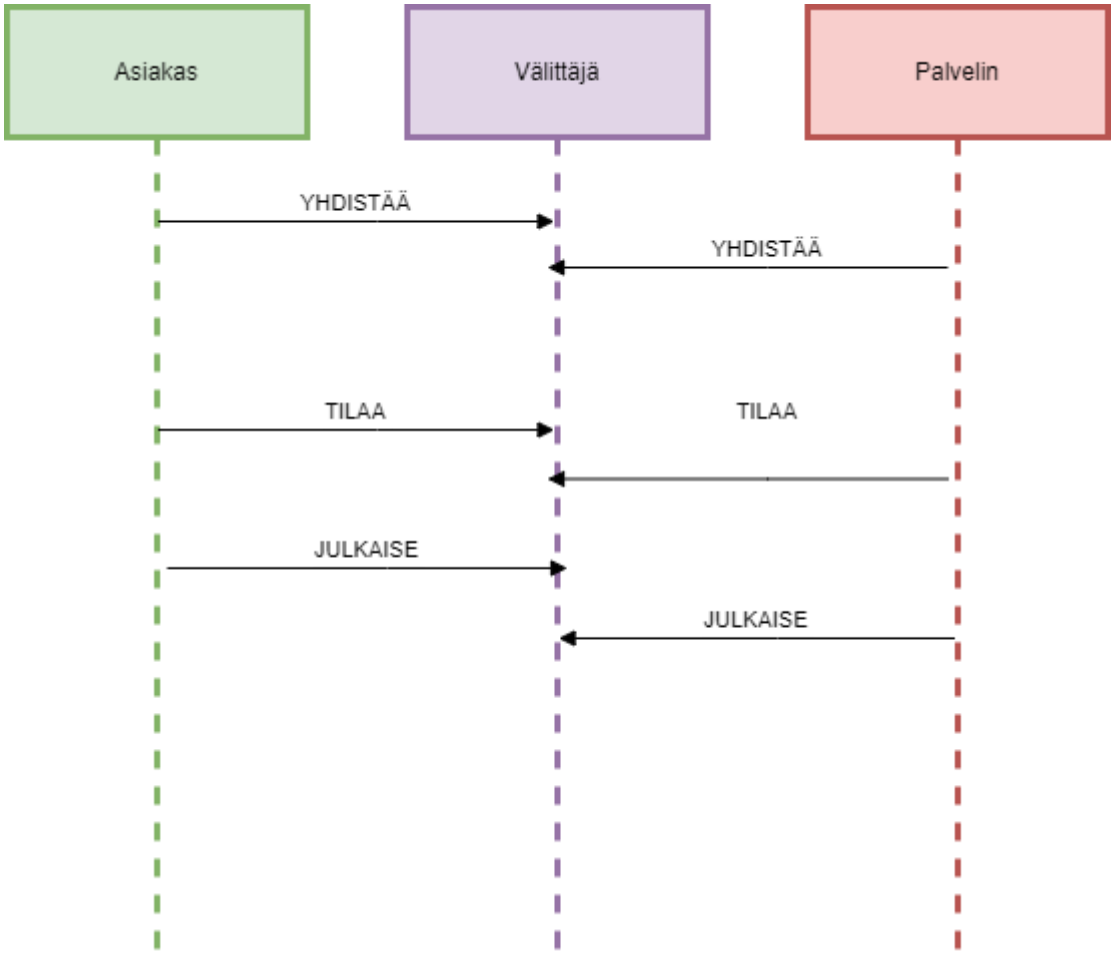
MQTT:n rakenne koostuu kolmesta pääkomponentista (Karagiannis ym., 2015, s.4):

1. **Asiakas:** MQTT-asiakas on ohjelma tai laite, joka muodostaa yhteyden MQTT-välittäjään ja julkaisee tai tilaa MQTT-aiheita. Asiakkaille voidaan kirjoittaa useilla ohjelmointikielillä ja niitä voidaan käyttää monenlaisilla laitteilla: anturit, sulautetut järjestelmät ja matkapuhelimet.

2. **Välittäjä:** MQTT-välittäjä on palvelin, joka vastaanottaa MQTT-asiakkaiden julkaisemia viestejä ja välittää ne muille asiakkaalle, jotka ovat tilannut asiaankuuluvat aiheet. Välittäjä vastaa asiakkaiden välisen yhteyden hallinnasta ja siitä, että viestit toimitetaan asianmukaisille tilaajille.

3. **Aihe:** MQTT-aihe on merkkijono, joka tunnistaa viestin sisällön ja sen vastaanottamisesta kiinnostuneet asiakkaat. Aiheet on järjestetty hierarkkiseen rakenteeseen, ja jokainen taso on erotettu vinoviivalla ("/"). Esimerkiksi "anturit/lämpötila/huone1" on aihe, joka ilmaisee lämpötila-anturin huoneessa 1. Asiakkaat voivat tilata tietyn aiheen tai aiheryhmän käyttämällä jokerimerkkejä, kuten "+" tai "#".

Ilman julkaisijatilaaja -rakennetta järjestelmän pitäisi kysyä jokaiselta laitteelta, onko hälytyksiä saatavilla. Tämä johtaisi väistämättä tarpeettomiin kyselyihin ja kuluttaisi IoT-laitteiden akkua. Siksi MQTT:n julkaisija ja tilaaja malli on ratkaisevan tärkeää tehokkaan ja toimivan viestinnän mahdollistamiseksi IoT-ympäristöissä. MQTT:n suunnitteluperiaatteet ja toiminnallisuus tekevät siitä suositun valinnan IoT-sovelluksiin, koska sen avulla laitteet voivat kommunikoida keskenään kevyesti ja tehokkaasti (Karagiannis, 2015, s. 4). Kuvio 9 kuvaa kommunikaatiota IoT-laitteen MQTT-asiakkaan, MQTT-välittäjän ja palvelimen MQTT-asiakkaan välillä.



Kuvio 9. MQTT-protokollan rakenne.

Näiden ydinkomponenttien lisäksi MQTT määrittää myös joukon viestityyppejä ja Quality of Service -tasoja, jotka määrittävät, kuinka viestit toimitetaan ja kuitataan asiakkaiden sekä välittäjien välillä. Viestityyppejä ovat Publish, Subscribe, Unsubscribe ja Ping/Response, kun taas QoS-tasot vaihtelevat QoS 0:sta, joka tarkoittaa korkeintaan kerran toimitettuna ja QoS 2:een, joka tarkoittaa kerran toimitus kuittauksella. Nämä ominaisuudet tekevät MQTT:stä joustavan ja luotettavan viestintäprotokollan IoT-sovelluksiin (Karagiannis ym., 2015, s. 4).

3.6 Mikrokontrolleri ja mittalaitteet

ESP32-mikrokontrolleri on suosittu valinta IoT-sovellusten kehittämisessä alhaisen virrankulutuksensa, pienen kokonsa ja sisäänrakennettujen langattomien ominaisuuksiensa vuoksi. Mikrokontrolleri voidaan ohjelmoida Arduino IDEllä. Arduino IDE on monipuolinen

ja käyttäjäystävällinen ohjelmistotyökalu, joka yksinkertaistaa ohjelmointiprosessia ja koodin lataamista mikrokontrollerikortille (Arduino, 2023). Arduino IDEssä on laaja kirjasto valmiina ja esimerkkejä eri antureille. Anturimittauksen mahdollistamiseksi ESP32-mikrokontrollerilla sen digitaalisiin ja analogisiin tuloihin voidaan kytkeä useita antureita. Tietoja näistä antureista voidaan kerätä ja lähettää langattomasti pilveen tai paikalliselle palvelimelle käyttämällä esimerkiksi Wi-Fiä (Espressif Systems, i.a.).

Anturimittaus on IoT-järjestelmien keskeinen ominaisuus ja sitä käytetään erilaisissa sovelluksissa (Buyya ym., 2016, s. 243–245). Esimerkiksi tällaisia on ympäristön valvonta, teollisuus automaatio, terveydenhuolto ja älykkäät kodit. Kirjoittajien mukaan anturit voidaan luokitella useisiin tyyppeihin niiden mitattavan fysikaalisen ilmiön, kuten lämpötilan, paineen, kosteuden, valon ja äänen perusteella.

3.7 Haavoittuvuudet ja niiden estäminen

IoT Security Foundation (i.a) tunnistaa seuraavat IoT-laitteiden haavoittuvuudet ohjelmisto-, laitteisto-, toimitusketju-, viestintä- ja määrityshaavoittuvuuksiin. IoT-laitteet käyttävät usein avoimen lähdekoodin ohjelmistoja, jotka voivat sisältää haavoittuvuuksia ja hyökkääjät voivat hyödyntää näitä. IoT-laitteita valmistavat usein kolmannet osapuolet, jotka voivat tuoda haavoittuvuuksia valmistusprosessin aikana. IoT-laitteet ovat riippuvaisia langattomasta viestinnästä, jonka hyökkääjät voivat siepata tai häiritä. IoT-laitteissa voi olla oletusasetuksia, jotka ovat turvattomia tai ne voivat olla väärin määritettyjä, jolloin ne altistavat hyökkäyksille.

IoT-järjestelmien valmistajien ja kehittäjien tulee noudattaa turvallisia ohjelmistokehityskäytäntöjä mahdollisten tietoturva-aukkojen tunnistamiseksi ja korjaamiseksi (Buyya ym., 2016, s. 243–244). Näihin käytäntöihin kuuluvat uhkien mallintaminen, koodien tarkistukset ja penetraatiotestaukset. Uhkamallinnus sisältää IoT-järjestelmään kohdistuvien mahdollisten uhkien tunnistamisen. Kuhunkin uhkaan liittyvien riskien analysoinnin ja asianmukaisten vastatoimien suunnittelun sekä näiden riskien vähentämiseksi. Tämä prosessi tulisi suorittaa ohjelmistokehityksen elinkaaren varhaisessa vaiheessa. Tällöin varmistetaan tietoturvan huomiointi prosessin jokaisessa vaiheessa. Kooditarkistuksiin kuuluu IoT-järjestelmän lähdekoodin tutkiminen haavoittuvuuksien ja koodausvirheiden tunnistamiseksi, joita

hyökkääjät voivat hyödyntää. Tämän prosessin tulee suorittaa koulutetut ammattilaiset, jotka tuntevat turvalliset koodauskäytännöt ja yleiset hyökkäystavat.

Läpäisytestauksessa simuloidaan tosielämän hyökkäyksiä IoT-järjestelmään haavoittuvuuksien ja heikkouksien tunnistamiseksi, joita hyökkääjät voivat hyödyntää (Buyya ym., 2016, s. 245). Tämän prosessin tulisi suorittaa kokeneet tietoturva-ammattilaisen, joka pystyy tunnistamaan mahdolliset hyökkäystavat ja arvioida olemassa olevien turvatoimien tehokkuutta. Näitä turvallisia ohjelmistokehityskäytäntöjä noudattamalla IoT-järjestelmien valmistajat ja kehittäjät voivat tunnistaa mahdollisia tietoturva-aukkoja ja korjata ne ennen kuin hyökkääjät voivat hyödyntää niitä.

3.8 Käyttäjien ja laitteiden hallinta

Azure AD on yksi vaihtoehto hallita käyttäjiä ja laitteita. Azure Active Directory (Azure AD) on Microsoftin tarjoama Identity and Access Management (IAM) -palvelu, joka tarjoaa laajan valikoiman ominaisuuksia. Vaikka sitä ei välttämättä pidetä parhaana IAM-palveluna kaikissa skenaarioissa, se kilpailee voimakkaasti muiden huippuvaihtoehtojen kanssa. Azure AD:n vertailu vaihtoehtoihin:

1. **Okta:** Okta on IAM-palvelu, joka keskittyy vahvasti pilvipohjaisen identiteettialustan tarjoamiseen. Okta on suosittu kilpailija Azure AD:lle, mutta Azure AD:n saumaton integrointi Microsoftin palveluihin tekee siitä houkuttelevamman organisaatioille, jotka käyttävät jo Microsoftin ekosysteemin tuotteita (CyberArk, i.a.).

2. **Ping Identity:** Ping Identity tunnetaan identiteetin ja pääsynhallintaratkaisuistaan. Siinä on Azure AD:ta vastaava ominaisuusvalikoima ja maailmanlaajuinen kattavuus (Ping Identity, i.a.).

3. **OneLogin:** OneLogin on IAM-palvelu, jota organisaatiot usein harkitsevat. OneLogin ei sisällä yhtä laajaa tietoturvaa kuin Azure AD (OneLogin, i.a.).

Azure AD on paras vaihtoehto, kun integroidaan ja käytetään Microsoft Azuren tuotteita jo valmiiksi. Parhaan IAM-palvelun valinta riippuu organisaation erityistarpeista ja käytössä

olevista palveluista. Azure AD tarjoaa useita keskeisiä ominaisuuksia, kuten identiteettien hallinnan, SSO ja sovellusten käyttöoikeuksien hallinta sekä laitehallintaa (Microsoftin, i.a. -b). Azure AD:n avulla organisaatiot voivat hallita käyttäjiä keskitetysti. Azure AD:n avulla käyttäjät voivat kirjautua sisään kerran omilla käyttöoikeustiedoillaan ja käyttää useita sovelluksia sekä resursseja ilman, että heidän tarvitsee syöttää tunnistetietojaan uudelleen. Tämä parantaa käyttäjien tuottavuutta ja vähentää useiden salasanojen hallinnan taakkaa. Azure AD tarjoaa myös laitehallintaominaisuuksia, joiden avulla organisaatiot voivat hallita ja suojata resurssejaan käyttäviä käyttäjälaitteita. Tämä sisältää ominaisuuksia, kuten laitteen rekisteröinnin, laitteen vaatimustenmukaisuuskäytännöt ja laitekohtaiset ehdolliset käyttökäytännöt.

Azure AD integroituu useisiin muihin Microsoftin pilvipalveluihin, kuten Office 365een, Dynamics 365een, Azureen ja kolmansien osapuolien sovelluksiin sekä palveluihin (Microsoft, i.a. -b). Azure AD tarjoaa sovellusliittymiä ja SDK:n eli kokoelma ohjelmistokehityksen työkaluja yhdessä asennettavassa paketissa kehittäjille, jotka voivat integroida Azure AD -todennuksen sovelluksiinsa.

3.9 OpenSSL-salaus

OpenSSL on laajalti käytetty avoimen lähdekoodin ohjelmistokirjasto, joka toteuttaa SSL- ja TLS-protokollia turvallisen tiedonsiirron tarjoamiseksi internetissä (OpenSSL, i.a.). Se tarjoaa kehittäjille seuraavat salaustoiminnot: salaus, salauksen purku, digitaalisen allekirjoituksen luominen ja vahvistaminen sekä varmenteiden hallinta. OpenSSL:ää käytetään esimerkiksi seuraavissa sovelluksissa, kuten web-palvelimet, sähköpostipalvelimet ja VPN.

OpenSSL:n toiminnallisuus on jaettu kolmeen pääkomponenttiin (OpenSSL, i.a.):

- **SSL/TLS-protokolla:** OpenSSL tarjoaa kattavan SSL/TLS-protokollan toteutuksen, joka mahdollistaa suojatun viestinnän asiakas- ja palvelinsovellusten välillä internetin kautta. Toteutus sisältää tuen uusimmille SSL/TLS-versioille sekä erilaisille salausohjelmistoille ja avaintenvaihtoalgoritmeille.

- **Salauskirjasto:** OpenSSL sisältää tehokkaan salauskirjaston, joka tukee eri salausalgoritmeja, kuten symmetrinen ja epäsymmetrinen salaus, digitaaliset allekirjoitukset, hash-funktiot ja satunnaislukujen luominen. Kirjasto on myös laajennettavissa, joten kehittäjät voivat lisätä uusia algoritmeja ja protokollia tarpeen mukaan.

- **Komentoriviohjelma:** OpenSSL sisältää komentorivityökalun, joka tarjoaa erilaisia työkaluja SSL/TLS-sertifikaattien ja -avaimien käsittelyyn sekä eri salaustoimintojen suorittamiseen. Apuohjelmaa voidaan käyttää SSL/TLS-sertifikaattien luomiseen, allekirjoittamiseen ja tarkistamiseen sekä tietojen salaamiseen ja salauksen purkamiseen käyttämällä eri algoritmeja.

OpenSSL-projektissa on toteutettu erilaisia toimenpiteitä kirjaston turvallisuuden parantamiseksi, kuten säännölliset tietoturvatarkastukset, bugipalkkio-ohjelmat ja yhteisön osallistuminen koodin tarkistamiseen ja testaukseen (OpenSSL, i.a.). Lisäksi monet organisaatiot ovat kehittäneet työkaluja ja parhaita käytäntöjä, jotka auttavat suojaamaan OpenSSL-asennuksia, kuten pitämään kirjaston ajan tasalla uusimmilla tietoturvakorjauksilla ja -kokoonpanoilla.

3.10 Azuren pilvipalvelu alustana palvelulle

Azure Cloud Services on kokoelma pilvipalveluita, joita Microsoft tarjoaa Azure-alustan kautta (Microsoft, i.a. -a). Azure tarjoaa myös laajan valikoiman palveluita, kuten virtuaalikoneita, tallennustilaa, verkottumista ja tietokantoja sekä erikoispalveluina analytiikka-, tekoäly- ja IoT-sovelluksiin. Azure-alusta on suunniteltu joustavaksi, skaalautuvaksi ja se tarjoaa yrityksille mahdollisuuden valita tarvitsemansa palvelut ja skaalata ylös tai alas tarpeen mukaan. Alusta tarjoaa kehittäjille myös valikoiman työkaluja ja resursseja sovellusten rakentamiseen, testaamiseen ja käyttöönottoon.

Platform as a Service (PaaS) on pilvilaskentamalli, jonka avulla kehittäjät voivat rakentaa ja ottaa käyttöön sovelluksia nopeasti sekä helposti ilman huolta taustalla olevasta infrastruktuurista (Microsoft, i.a. -c). PaaS:ssä pilvipalveluntarjoaja tarjoaa alustan, joka koostuu työkaluista, kirjastoista ja muista resursseista, joita tarvitaan sovellusten kehittämiseen, testaamiseen ja käyttöönottoon. PaaS tarjoaa myös automaattisen skaalautuvuuden,

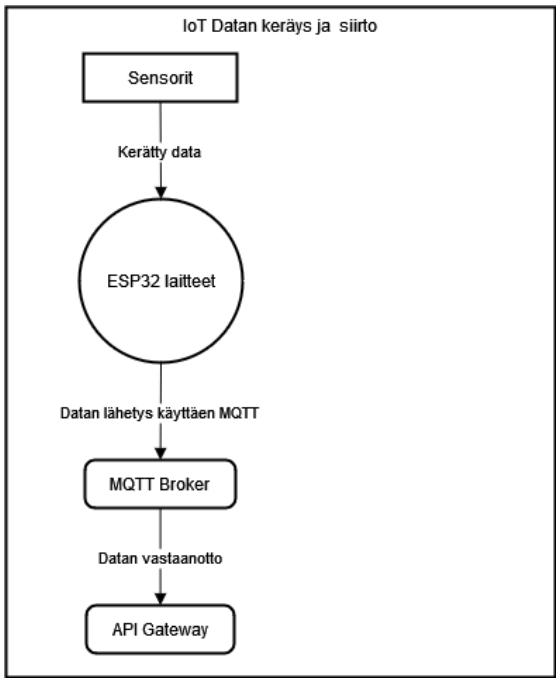
kuormituksen tasapainotuksen ja muita ominaisuuksia, jotka yksinkertaistavat sovellusten käyttöönottoa ja hallintaa. PaaS antaa kehittäjille mahdollisuuden keskittyä sovellusten rakentamiseen ja parantamiseen taustalla olevan infrastruktuurin hallinnan sijaan. Tämä voi nopeuttaa markkinoille tuloa ja alentaa kustannuksia sekä parantaa skaalautuvuutta ja luotettavuutta.

PaaS:llä on joitain ongelmia, kuten toimittajan lukitus sekä taustalla olevan infrastruktuurin rajoitettu hallinta ja mahdolliset tietoturvariskit (Kavis, 2014, s. 106–107). Toimittajan lukituksella tarkoitetaan tilannetta, jossa käyttäjä tulee riippuvaiseksi tietystä PaaS-palveluntarjoajasta ja hänen on vaikea siirtyä toiselle palveluntarjoajalle. Perusinfrastruktuurin rajallinen hallinta tarkoittaa, että käyttäjillä ei ehkä ole pääsyä sovellusten suorittamiseen käytettyihin laitteisto- ja ohjelmistoresursseihin, mikä vaikeuttaa ympäristön mukauttamista tai optimointia. Lopuksi PaaS:ään liittyviä turvallisuusriskejä ovat tietomurrot, luvaton pääsy ja palveluntarjoajan mahdollisuus paljastaa käyttäjätietoja.

4 PROTOTYYPIN JÄRJESTELMÄARKKITEHTUURI

4.1 Laitekerros

Laitekerroksessa ovat IoT-laitteet, jotka keräävät ja lähettävät dataa verkkoon. Tässä käytetään ESP32:ta lämpötila- ja kosteustietojen keräämiseen ja lähettämiseen MQTT:n avulla. IoT-laitteiden keräämät tiedot siirretään MQTT-välittäjälle MQTT-protokollan kautta, mikä varmistaa kevyen ja tehokkaan tiedonsiirron. MQTT-välittäjä vastaanottaa tiedot ja välittää sen API-yhdyskäytävälle, joka toimii rajapintana IoT-verkon ja käyttöliittymän välillä. MQTT tarjoaa julkaisu ja tilaa -mallin, jossa laitteet julkaisevat dataa aiheeseen ja tilatut asiakkaat voivat vastaanottaa tiedot. Tässä arkkitehtuurissa käytetty MQTT-välittäjää isännöi Azure Virtual Machines ja data liikkuu JSON-formaatissa IoT-yhdysväylässä sekä MQTT-viestit käyttävät JSON-formaattia. Kuviossa 10 on esitetty tiedonsiirto alimmassa kerroksessa.



Kuvio 10. IoT-Datan keräys ja siirto.

Mikro-ohjaimena käytetään ESP32:ta. Valitussa dev-boardissa on sisäänrakennettuna WiFi-ominaisuus, jonka avulla voidaan muodostaa yhteys internetiin ja kommunikoida muiden laitteiden kanssa. IoT-sensorin suunnitteleminen ESP32:ta ja WiFiä käyttäen on

valittava sopiva anturi, joka pystyy keräämään tietoa ympäristöstä. Kun anturi on valittu, se liitetään ESP32:een ja kirjoitetaan ohjelma, joka lukee anturin tiedot ja lähettää ne WiFin kautta käyttäen MQTT-protokollaa. Seuraavaksi asetetaan ESP32 muodostamaan yhteys MQTT-välittäjään WiFi-verkon avulla. ESP32 voi lähettää oikeassa muodossa olevia tietoja välittäjälle MQTT-protokollan avulla. MQTT on kevyt protokolla, jonka julkaisu-/tilausmallin avulla laitteet voivat tilata tiettyjä aiheita ja vastaanottaa vain niille tärkeitä tietoja. MQTT-välittäjät voivat käsitellä suuria tietomääriä ja ne voidaan konfiguroida toimimaan useiden laitteiden kanssa samanaikaisesti.

ESP32:een toteutetaan oma pieni käyttöjärjestelmä, johon pystytään käyttäjän toimesta määrittelemään asetuksia. Samalla näille asetuksille toteutetaan tallennus ja talteenotto. Näin pystytään tallentamaan laitteeseen I/O-asetukset, verkkoasetukset, salausavaimet ja MQTT-yhteysasetukset. Nämä tiedot tallennetaan JSON-formaatissa laitteen pysyväismuistiin, joka säilyy virran katkaisun yli. Tämä helpottaa IoT-laitteen asetusten asettamista, kun siihen voidaan muodostaa yhteys laitteella, jossa on WiFi ja verkkoselain. Esimerkkinä tällainen laite voi olla matkapuhelin tai kannettavalla tietokone. Mikrokontrollerin oma käyttöjärjestelmä mittaa tietoa, jonka se muuntaa selkolukuiseksi ja syöttää MQTT-viestiksi. Laitteelle tulee myös käskyjä MQTT-protokollaa pitkin mitata ja pysäyttää mittaus sekä ohjata ja lukea valittuja uloslähtöjä sekä sisääntuloja. MQTT-viestit salataan käyttäen OpenSSL-salausta.

4.2 Mikropalvelukerros

Keskikerroksen muodostavat Azure-pilviympäristön virtuaalikoneet tai tarvittaessa muut virtuaalikoneet, jotka vastaavat IoT-järjestelmän muodostavien mikropalvelujen suorittamisesta. Nämä virtuaalikoneet tarjoavat tarvittavat laskentaresurssit, kuten prosessointitehon, muistin ja tallennustilan, jotta palvelut toimivat tehokkaasti. Tässä kerroksessa toimivat mikropalveluina SQL-tietokanta, MQTT-välittäjä, API-mikropalvelu ja näitä palveluita ohjaava sekä kontrolloiva logiikkapalvelu.

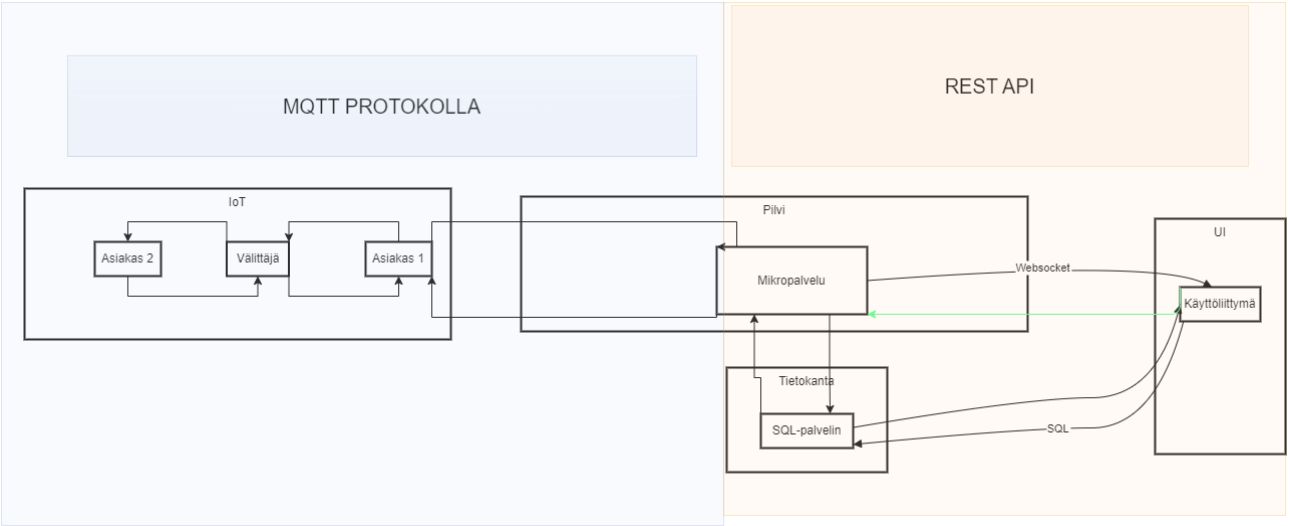
Keskikerros vastaa alemmalta kerrokselta vastaanotettujen tietojen käsittelystä ja tallentamisesta. MQTT-välittäjä vastaanottaa tiedot ja tallentaa ne SQL-tietokantaan, jota isännöi Azure Virtual Machines. Azure Virtual Machines -ympäristössä isännöidyt mikropalvelut

voivat hyödyntää SQL-tietokantaan tallennettuja tietoja käyttäen REST API -kutsuja. Tässä suunnitteluvaiheessa REST API:n käyttäminen vaikuttaa tähän käyttötarkoitukseen sopivalta. MQTT-protokollan käyttämiseksi on valittava MQTT-välittäjä, joka voi vastaanottaa tietoja ESP32:sta ja välittää ne pilveen. Välittäjä voidaan isännöidä Azuren pilvialustalla, mutta sen pitää pystyä pystyttämään myös Docker-imagena.

4.2.1 REST API ja MQTT-kommunikointi

REST API on kriittinen komponentti IoT-verkon ja mikropalveluiden välisessä kommunikoinnissa. Se toimii välittäjänä etukäyttöliittymän ja palveluita isännöivien virtuaalikoneiden välillä. API-yhdyskäytävä vastaa etukäyttöliittymän tekemien pyyntöjen käsittelystä ja niiden reitittämisestä sopivalle mikropalvelulle, joka suorittaa sen lähettämän pyynnön. Suunnitellessa API-yhdyskäytävä tällaiselle arkkitehtuurille on ensin määritettävä yhdyskäytävän vaatimukset ja toiminnot. Yhdyskäytävän tulisi tarjota kehittäjille yksinkertainen ja yhtenäinen käyttöliittymä IoT-verkon tietoihin. Sen tulee olla skaalautuva ja vikasietoinen, jotta se pystyy käsittelemään suuria määriä käyttäjien pyyntöjä. Yhdyskäytävän tulee myös tarjota turvatoimia luvattoman käytön estämiseksi ja varmistaa IoT-laitteiden sekä virtuaalikoneiden välillä siirrettävän tiedon turvallisuus.

API-yhdyskäytävän luomiseksi käytetään omaa mikropalvelua, joka tulkitsee API-komennossa tulevan komennot ja muuntaa sen MQTT-käskyksi ja lähettää MQTT-välittäjäpalvelulle. Samainen API-palveluita käsittelevä palvelu voidaan koodata mahdollistamaan pääsy SQL-tietokantaan tallennettuihin tietoihin ja palauttamaan ne suoraa käyttöliittymälle. MQTT-asiakaspalvelu osuus voidaan määrittää kuuntelemaan saapuvia viestejä MQTT-välittäjältä ja näissä viesteissä saapuva data voidaan lähettää tallennettavaksi tietokantaan. Tiedonkäsittely ja hallinnoiva mikropalvelu voi käsitellä saapuvat pyynnöt, tehdä kyselyitä SQL-tietokannasta ja lähettää vastauksen takaisin MQTT-asiakaspalveluun. MQTT-asiakaspalvelu voi sitten välittää vastauksen takaisin ESP32-laitteelle. Kuviossa 11 on MQTT ja API-kommunikointien alueet eriteltynä arkkitehtuurissa.



Kuvio 11. Esimerkki MQTT-viestiliikenteestä ja REST API-komennoista.

MQTT-käytävän tarkoitus on mahdollistaa tietoliikenne IoT-laitteiden ja järjestelmän välillä, johon ne ovat yhteydessä. Toisin sanoen MQTT toimii siltana ja välittäjänä laitteiden sekä järjestelmän välillä. IoT-laitteet luovat dataa eri muodoissa, jotka eivät välttämättä ole yhteensopivia järjestelmän käyttämän muodon kanssa. MQTT-protokolla vastaa laitteiden lähettämien tietojen muuntamisesta järjestelmän käsittelemään muotoon. Tämän muunnoksen avulla järjestelmä voi vastaanottaa ja käsitellä tietoja laitteista. Tietojen muuntamisen lisäksi MQTT-protokollaan lisäämällä SSL-salaus huolehtii tietoturvasta ja salauksesta laitteiden sekä järjestelmän välisessä viestinnässä. Näin varmistetaan laitteiden lähettämien tietojen turvallisuus ja luvattomat eivät pääse niihin käsiksi.

4.2.2 Virtuaalipalvelin ja pilvipalvelu

Keskeisiä vaatimuksia on skaalautuvuus, luotettavuus, turvallisuus ja helppokäyttöisyys kehittäjille. Tärkeä näkökohta IoT-arkkitehtuurin PaaSn suunnittelussa on sopivan pilviinfrastruktuurin valinta. Tähän sisältyy pilvipalveluntarjoajan valitseminen, joka tarjoaa tarvittavat resurssit ja palvelut sopivaan hintaan. Infrastruktuurin pitää pystyä käsittelemään IoT-laitteiden tuottaman suuren datamäärän ja skaalaamaan käsittelykykyä tarpeen mukaan. Tärkeänä lähtökohtana on väliohjelmisto- ja sovelluspalveluiden valinta, joita käytetään IoT-sovellusten rakentamiseen ja käyttöönottoon PaaS:ssä. Tämä voi sisältää tietokantoja, viestivälittäjiä ja virtualisointi työkaluja. Palveluiden tulee olla suoraan yhteensopivia keskenään ja mieluusti toisiaan suoraa tukevia. Tärkeää on suunnitella virtuaalipalvelut

ja pilvipalvelut turvallisuus mielessä. Tämä tarkoittaa todennus- ja valtuutusmekanismien käyttöönottoa kaikkien tietojen suojaamiseksi luvattomalta käytöltä. Tämä sisältää salausta ja muita toimenpiteitä tiedonsiirron turvaamiseksi laitteiden sekä pilven välillä. Tähän voi myös sisällyttää valvonta- ja hälytysmekanismien toteuttamista tietoturvauhkien havaitsemiseksi ja niihin reagoimiseksi reaaliajassa. Prototyypissä pilvipalvelusta lähtevä kommunikointi IoT-laitteelle salataan käyttäen OpenSSL-salausta. Pilvipalvelu tulee olemaan Azure AD-kirjautumisen takana, koska käytetään Azuren pilvipalvelua. Azuren pilvipalvelusta löytyy hallintatyökaluista lokia käyttäjistä, palveluiden toiminnoista ja muodostetuista yhteyksistä.

Azure-pilvivirtuaalikonepalveluita voidaan käyttää tukemaan IoT-arkkitehtuuria. Azurelta löytyy oma IoT HUB-palvelu kokonaisuus. Prototyypissä sitä ei käytetä, koska ei haluta jäädä yhden toimittajan loukkuun ja nämä palvelut on tarvittaessa pystyttävä itse pystyttämään omassa palvelussa tai omalla virtuaalikoneella. IoT:n yhteydessä virtuaalikoneilla voidaan ajaa IoT-ratkaisun eri komponentteja, kuten IoT-yhdyskäytävää, tietokantapalvelinta ja sovelluspalvelinta.

Aluksi Azuressa luodaan virtuaalinen verkko, joka sisältää aliverkot toteutuksen eri komponenteille. Näin luodaan aliverkko IoT-käytävälle, tietokantapalvelimelle ja sovelluspalvelimelle. Seuraavaksi luodaan virtuaalikoneen IoT:lle ja asentaa siihen tarvittavat ohjelmistot, kuten MQTT-välittäjän. Tämän jälkeen voidaan määrittää virtuaalikone muodostamaan yhteyden IoT-laitteisiin ja lähettämään tiedot pilveen. Virtuaalikone voidaan konfiguroida skaalaamaan ylös tai alas käsiteltävien laitteiden määrän perusteella, mikä voi auttaa varmistamaan, että IoT-ratkaisu pysyy reagoivana myös laitteiden määrän kasvaessa. Sama voidaan tehdä virtuaalikoneilla olevalle tietokantapalvelimelle ja sovelluspalvelimelle. Tietokantapalvelin isännöi SQL-tietokantaa, joka tallentaa IoT-laitteiden tiedot, kun taas sovelluspalvelin voi isännöidä verkkosovellusta tai muuta ohjelmistoa, joka käsittelee tietoja.

Korkean käytettävyyden ja vikasietoisuuden varmistamiseksi määritellään virtuaalikoneet käyttämään Azure-saatavuusjoukkoja. Tämä mahdollistaa virtuaalikoneiden jakamisen useille vika-alueille, jolloin yhden virtuaalikoneen tai toimialueen vikaantuessa järjestelmä jatkaa toimintaansa. Kuormituksen tasapainottimet jakavat liikenteen useiden sovelluspalvelimien kesken, mikä varmistaa sovelluksen toiminnan myös ruuhkaliikenteen aikana.

Turvallisuuden kannalta voimme käyttää Azure Security Centeriä virtuaalikoneiden valvomiseen ja turvalliseen määrittelyyn. Azure Active Directoryllä hallitaan käyttäjien todennusta ja pääsyä IoT-järjestelmän eri komponentteihin.

4.2.3 Mikropalvelut Docker-imagena ja pilvipalvelussa

Prototyypissä paketoidaan Mosquito ja palvelinohjelma Docker-kontiksi. Tämä toimii testauksena Dockerin toiminnallisuudelle Azuren pilviympäristössä. Prototyypissä käännämme myös perinteisen applikaation. Tällöin pilvipalvelussa voimme ajaa virtuaalikoneella Linux tai Windows-ympäristössä palvelinohjelmistoa sekä Mosquitoa. Näin ei ole myöskään riippuvaisuutta Dockerista. Tällä pyritään estämään toimittajaloukku ja liiallinen riippuvuus Dockeriin.

Docker-kuvatiedostoa voidaan ajaa pilvipalvelussa, joka tukee Docker-kuvatiedostoja tai omalla palvelimella. Suunniteltu Docker-kuva voidaan ottaa käyttöön Azure-pilvipalvelussa ja Mosquito voidaan myös paketoida Docker-kuvaksi. Ensimmäinen askel Docker-kuvan suunnittelussa on valita peruskuva. Azure tarjoaa erilaisia peruskuvia eri käyttöjärjestelmille ja ohjelmointikielille. Tässä tapauksessa valitaan Linux-pohjainen kuva, joka sisältää tarvittavat riippuvuudet Mosquiton suorittamiseen. Sitten voidaan käyttää Docker-tiedostoa, jotta määritetään kuvan rakentamiseen tarvittavat vaiheet, mukaan lukien kaikki lisäpaketit tai kokoonpanot, joita tarvitaan Mosquiton toimimiseen oikein.

Paketoitu Docker-kuva voidaan ottaa käyttöön Azure-pilvipalveluissa Azure Container Registryn tai muun kontin organisointipalvelun avulla. Docker-kontin käytöstä on hyötyä prototyypin arkkitehtuurissa, koska sen avulla on mm. mahdollista eristää Mosquiton muista samassa isäntäkoneessa toimivista mikropalveluista. Tämä auttaa varmistamaan, että Mosquiton ja muiden mikropalveluiden ongelmat tai tietoturvahaavoittuvuudet eivät vaikuta arkkitehtuurimme muihin osiin. Käyttämällä konttilähestymistapaa voidaan helposti siirtää mikropalveluita eri ympäristöjen, kuten kehitys-, testaus- ja tuotantoympäristöjen välillä.

4.2.4 Tietokantapalvelu

SQL-tietokantaa käytetään Azure-pilvipalvelussa tietojen tallentamiseksi IoT-anturiverkosta ja siihen on määritettävä SQL-tietokannan rakenne ja skeema. Prototyypissä tietokantaan tallentamaan IoT-anturiverkon tuottamaa dataa. Tietokanta rakentuu organisaatiotaulusta, jonka alle kerätään organisaatioon liitettyjen sijaintien anturitaulukkoon mittaustietoa. Huomioon otettavaa on SQL-tietokannan turvallisuus, koska IoT-anturiverkko kerää liiketoiminnallista dataa ja välittää käskyjä. Prototyypissä tietokantaan on määritelty käyttäjät eri oikeuksilla, jolloin voidaan seurata lokeista käyttöä ja käyttäjiä. Azure SQL Database on kryptausominaisuudet. Tällöin on varmistettava, että tiedot on suojattu tallennettuna.

Huomioitavaa on myös IoT-anturiverkon tietojen käyttö ja käsittely. Prototyypissä pyritään käyttämään yksinkertaisia rakenteita. Tämä tarkoittaa normaalikäytössä yhden rivin tai taulun päivittämistä pienellä tiedon määrällä. Isommat ja usean taulun haut ja tallennukset tapahtuisivat proseduurikutsuilla. Pyrkimyksenä on välttää turhia SQL-tietokantakyselyitä sekä liiallisen tiedon turhaan hakemista. On otettava huomioon SQL-tietokannan skaalautuvuus ja saatavuus. IoT-anturiverkon kasvaessa tietokannan on kyettävä käsittelemään kasvavaa tietomäärää ja pyyntöjä suorituskyvystä tinkimättä. Azure SQL Databasesta löytyy automaattinen skaalaus tietokantaan varattujen resurssien säätämiseen tarpeen mukaan. Tässä on kumminkin otettava huomioon kasvava kustannus, koska Azuren tarjoamat lisäresurssit eivät ole ilmaisia. Prototyypissä on luontevinta seurata Azuressa käytettyjen resurssien reaaliaikaseurantaa ja suorittaa suorituskykytesti järjestelmän toteutukselle.

4.3 Käyttöliittymäkerros

Yläkerros tarjoaa käyttöliittymän ja pääsyn keskikerrokseen tallennettuihin tietoihin APIn kautta. Asiakkaat voivat hakea tietoja SQL-tietokannasta APIn avulla. API vastaa pyyntöjen reitittämisestä asianmukaisiin mikropalveluihin. API-käytävä käyttää Azure AD todennus- ja valtuutusominaisuuksia varmistaakseen, että vain valtuutetut käyttäjät voivat käyttää sovellusliittymä. Näin käyttäjille saadaan yhtenäinen ja helppokäyttöinen käyttöliittymä, jossa mikropalvelut liitetään käyttöliittymään API-käytävän avulla. Tämä API-käytävä

tarjoaa yksinkertaisen REST-sovellusliittymän, jonka avulla kehittäjät voivat käyttää IoT-verkon tietoja.

Käyttöliittymään kirjaudutaan käyttäen Azure AD-kirjautumista eli kansankielisesti Microsoftin omaa sisäänkirjautumisen SSO-tekniikkaa. Käyttöliittymässä pääsee käsiksi mittadataan, jota IoT-verkosta tulee. Tätä tietoa pystyy graafisesti tutkailemaan ja suodattamaan muutamalla suodatusehdolla. Käyttöliittymästä pystytään myös lähettämään käskyjä IoT-laitteille, jolloin voidaan keskeyttää mittaustieto tai ohjata mikrokontrollerin lähtöjä. Käyttöliittymä on myös pystyttävä ajamaan omana Docker-kuvana tai ohjelmana virtuaalikoneella, jolloin se voidaan pystyttää pilveen tai omalle palvelimelle.

4.4 Dokumentaation suunnittelu & toteutus

Ohjelmistodokumentaatio on tärkeä osa ohjelmistokehitysprojektia. Asianmukainen dokumentaatio varmistaa, että järjestelmää voidaan helposti ylläpitää, ymmärtää ja päivittää. Asiakirjat tulee kirjoittaa selkeästi ja ytimekkäästi käyttäen yksinkertaista kieltä, jota kaikki sidosryhmät ymmärtävät. Ohjelmistodokumentaatiossa tulee olla kuvaus järjestelmän muodostavista laitteisto- ja ohjelmistokomponenteista. Tämä sisältää IoT-laitteet, yhdyskäytävät, pilvipalvelut ja muut infrastruktuurikomponentit. Dokumentaation tulee selittää, miten kukin komponentti on kytketty ja miten data kulkee järjestelmän läpi. Tiedonhallinta on tärkeä osa ohjelmistodokumentaatiota. Asiakirjoissa tulee selittää, kuinka tietoja kerätään, tallennetaan, käsitellään ja analysoidaan järjestelmässä. Sen tulee kuvata järjestelmässä käytetyt tietorakenteet ja kuinka data muuntuu sen kulkiessa eri komponenttien läpi. Lisäksi dokumentaatiosta on löydyttävä, kuinka tiedot suojataan, varmuuskopioidaan ja arkistoidaan.

Testaus ja virheenkorjaus ovat myös tärkeitä ohjelmistodokumentaation näkökohtia. Dokumentaatiossa tulee kuvata, miten järjestelmää testattiin kehityksen aikana ja kuinka sitä ylläpidetään tai päivitetään. Dokumentaatio sisältää kuvaukset testausympäristöistä, testidatasta ja testausmenetelmistä. Dokumentaation tulee antaa ohjeita järjestelmän käyttöön ja ylläpitoon. Käyttö ja ylläpito dokumentointi sisältää tietoja järjestelmän asentamisesta ja määrittämisestä, järjestelmän suorituskyvyn valvonnasta ja mahdollisten ongelmien vianmäärityksestä.

4.5 Tietoturvan suunnittelu, valvonta ja testaus

Ensimmäinen vaihe on ottaa käyttöön suojattu viestintäprotokolla, kuten OpenSSL salataksi tiedonsiirron IoT-laitteiden ja pilvipalvelimien välillä. Tämä auttaa estämään siirrettävien tietojen luvattoman käytön tai muuttamisen. Turvallisen viestinnän lisäksi tulisi ottaa käyttöön kulunvalvontatoimenpiteitä, joilla rajoitetaan pääsyä IoT-verkkoresursseihin. Tämä sisältää käyttäjän todennuksen ja roolipohjaisen pääsynhallinnan toteuttamisen pilviresurssien käytön rajoittamiseksi käyttäjäroolien perusteella.

Tärkeä IoT-turvallisuuden kohta on laitteiden todennus ja valtuutus. IoT-laitteet tulee näkyä pilvipalvellussa yksilöllisellä tunnisteella. Näin tiedetään laitteet, jotka ovat muodostaneet yhteyden pilvipalveluihin ja lähettäneet dataa. Pilvipalveluihin tallennetun tiedon turvaamiseksi tulee ottaa käyttöön tiedon salaus- ja varmuuskopiointimekanismit. Arkaluonteisettiedot tulee salata ja varmuuskopioida säännöllisesti, jotta estetään tietojen menetys järjestelmävikojen tai tietoturvaloukkausten sattuessa. Edellä mainittujen turvatoimenpiteiden lisäksi järjestelmän haavoittuvuuksien tunnistamiseksi tulee perehtyä yleisiin tietoturvauhkien, kuten injektiohyökkäysten, sivustojen välisten komentosarjojen hyökkäysten ja muihin yleisiin hyökkäystapoihin. Perehtymisen jälkeen pyritään kehityksessä tiedostamaan uhat ja mahdollisesti miettimään puolustautumiskeinoja.

Tietoturvan valvonnassa on hyvä perehtyä jatkuvan seurantatyökaluihin, jotka voivat tunnistaa epätavallisia verkkoliikennemalleja tai muuta epäilyttävää toimintaa. Näitä työkaluja voivat olla tunkeutumisen havaitsemis- ja estojärjestelmät, palomuurit sekä tietoturvatieto- ja tapahtumahallintajärjestelmät. Tarpeellista on myös säännöllisesti tarkistaa näiden valvontatyökalujen luomat lokit ja hälytykset mahdollisten tietoturvahäiriöiden tai haavoittuvuuksien tunnistamiseksi.

Eri tavat testata järjestelmän tietoturvaa:

1. **Läpäisytestaus:** Tässä simuloidaan järjestelmää vastaan tehtyä hyökkäystä haavoittuvuuksien ja heikkouksien tunnistamiseksi. Ammattimainen läpäisytestaaja voi suorittaa tämän testin ja toimittaa raportin tuloksista. Prototyypissä ei alkukehitysvaiheessa toteuteta tätä.

2. **Haavoittuvuuden tarkistus:** Tämä tarkoittaa automaattisten työkalujen käyttöä järjestelmän haavoittuvuuksien tarkistamiseksi. Nämä työkalut voivat tunnistaa yleisiä haavoittuvuuksia, kuten vanhentuneita ohjelmistoja tai heikkoja salasanoja. Prototyypin kehitysvaiheessa tarkkaillaan käytettyjen ohjelmistokirjastojen kriittisistä haavoittuvuuksista tulevia ilmoituksia. Salasanoja ei tallenneta koodiin tai versiohallintaan.

3. **Koodikatselmointi:** IoT-järjestelmän lähdekoodi voidaan tarkistaa mahdollisten tietoturvapuutteiden tunnistamiseksi. Tämä voidaan tehdä manuaalisesti tai automaattisten työkalujen avulla. Prototyypissä koodikatselmoinnin tekee yrityksen tähän opinnäytetyöhön liitetty yhteyshenkilö.

4. **Tietojen salauksen testaus:** Järjestelmässä siirrettävien tietojen turvallisuutta voidaan testata tarkistamalla, onko se salattu. Testaus voidaan tehdä yrittämällä siepata tiedonsiirtoja ja analysoimalla datapaketteja. Prototyypissä Azure SQL Databasen tulisi hoitaa tämä automaattisesti.

5. **Kulunvalvontatestaus:** Järjestelmän kulunvalvontamekanismeja voidaan testata yrittämällä ohittaa todennusprosessi ja päästä sisään rajoitetuille alueille. Prototyypissä tämä todetaan tarkistamalla lokit ja onko niihin jäänyt merkintöjä.

6. **Kuormitustestaus:** Tämä sisältää järjestelmän suuren liikenteen simuloinnin sen suorituskyvyn ja turvallisuuden testaamiseksi raskaan kuormituksen aikana. Prototyypissä ei tässä vaiheessa toteuteta tätä.

7. **Vaatimustenmukaisuustestaus:** Tämä sisältää testauksen, täyttääkö IoT-järjestelmä tietoturvastandardit ja -määräykset, kuten GDPR. Prototyypin toteutuksen tuloksissa katselmoidaan tietoturvastandardien vastaavuus.

8. **Disaster Recovery Testing:** Tämä sisältää järjestelmän varmuuskopiointi- ja palautusmenettelyjen testaamisen katastrofin tai järjestelmävian sattuessa. Prototyypissä tämä toteutetaan äkillisesti uudelleen käynnistämällä palvelu ja toteamalla mitä tapahtui.

5 PROTOTYYPIN TOTEUTUS JA TULOKSET

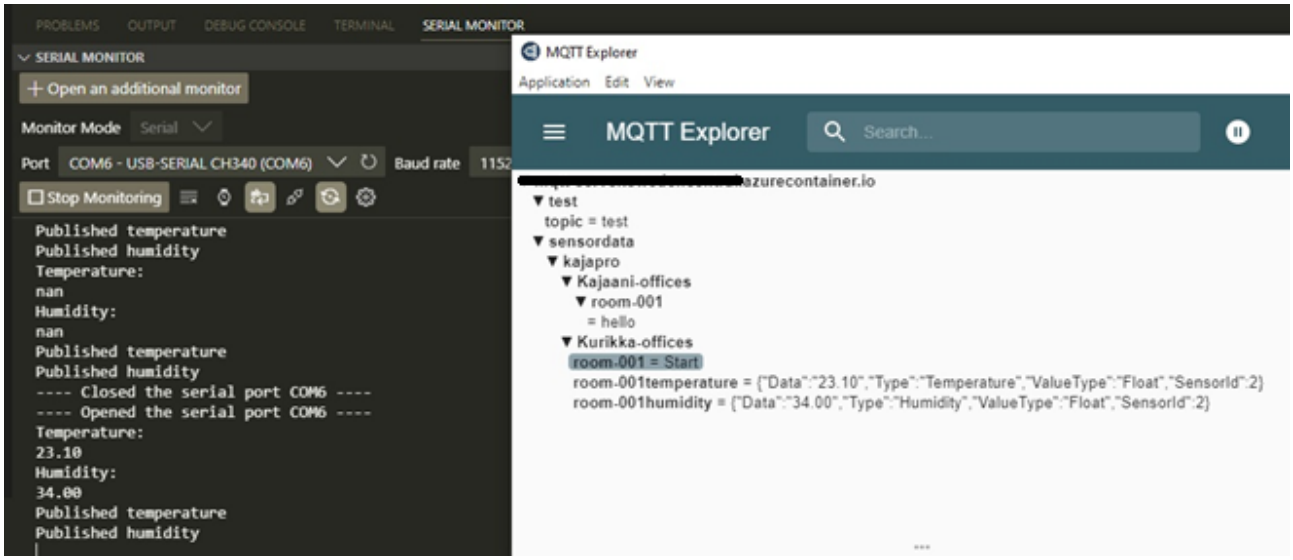
5.1 Mikrokontrolleri ja sensori

Kehitysympäristönä käytettiin Visual Studio Codea (Microsoft Corporation, 2023) ja siihen integroitiin PlatformIO (PlatformIO, 2023) sekä Arduino IDE (Arduino, 2023). Näin saatiin nopeasti pystytettävä kehitysympäristö avoimenlähdekoodinympäristöillä, joiden lisenssit mahdollistivat myös mahdollisen tuotteistamisen tulevaisuudessa. Kehitys aloitettiin Arduino IDE lisäosaa käyttäen, mutta toteutuksen edistyessä alkoi ilmetä epästabiilisuutta ja epämääräisiä ongelmia MQTT:tä käytettäessä. PlatformIO lisäosaa testattiin ja sitä käyttäessä näitä ongelmia ei esiintynyt, joten Arduino IDE-lisäosa vaihdettiin PlatformIO-lisäosaan. PlatformIO tarjosi kattavat kirjastot, kääntötyökalut ja mikrokontrollerilla ohjelmistokoodin siirtämisen. Nämä helpottivat ja nopeuttivat ohjelmistokehitystä. Visual Studio Code oli liitetty Git (Git Contributors, 2023) versiohallintaan ja versiohallinta oli Azure DevOpssin (Microsoft Corporation, i.a.) pilvessä.

Kuvassa 12 näkyy mikrokontrollerissa käytetty valmis wifimanageri-ohjelmistokirjasto, johon lisättiin tarvittavat lisäkentät MQTT-yhteystietojen talteenottoon ja tallentamiseen Flash-muistiin json-muodossa. Näin mikrokontrolleri asetusten asettamisvaiheessa näkyy WiFi-pisteenä, johon saadaan yhteys. Yhteyden muodostus ohjaa automaattisesti verkkoselainnäkymään. Verkkoselain näkymästä pystyi kuvan mukaisesti asettamaan WiFi-verkkoon, josta laite pystyy kommunikoimaan MQTT-viestinvälittäjälle, kun oikeat MQTT-yhteysasetukset on myös annettu. Kaikki viestintä on salattu käyttäen OpenSSL:ää.

Kuva 12. IoT-laitteen graafinen näkymä.

Kuvassa 13 on vasemmalla debug tarkoituksessa mikrokontrollerin sarjaporttiin kirjoittamat viestit, joita seuraamalla voidaan tarkistaa sekä validoida mikrokontrollerin ohjelmistokoodin toimivuus laitteessa. Oikeanpuoleisessa kuvassa on MQTT-explorer ohjelmisto liitettynä MQTT-välittäjään, jolloin voidaan tarkkailla eri laitteista tulevia viestejä MQTT-välittäjään ja varmistaa niiden oikeamuotoisuus. Tämä on visuaalisesti selkeämpi ja helpompi tapa kuin lukea suoraan MQTT-välittäjän lokitiedostoa.



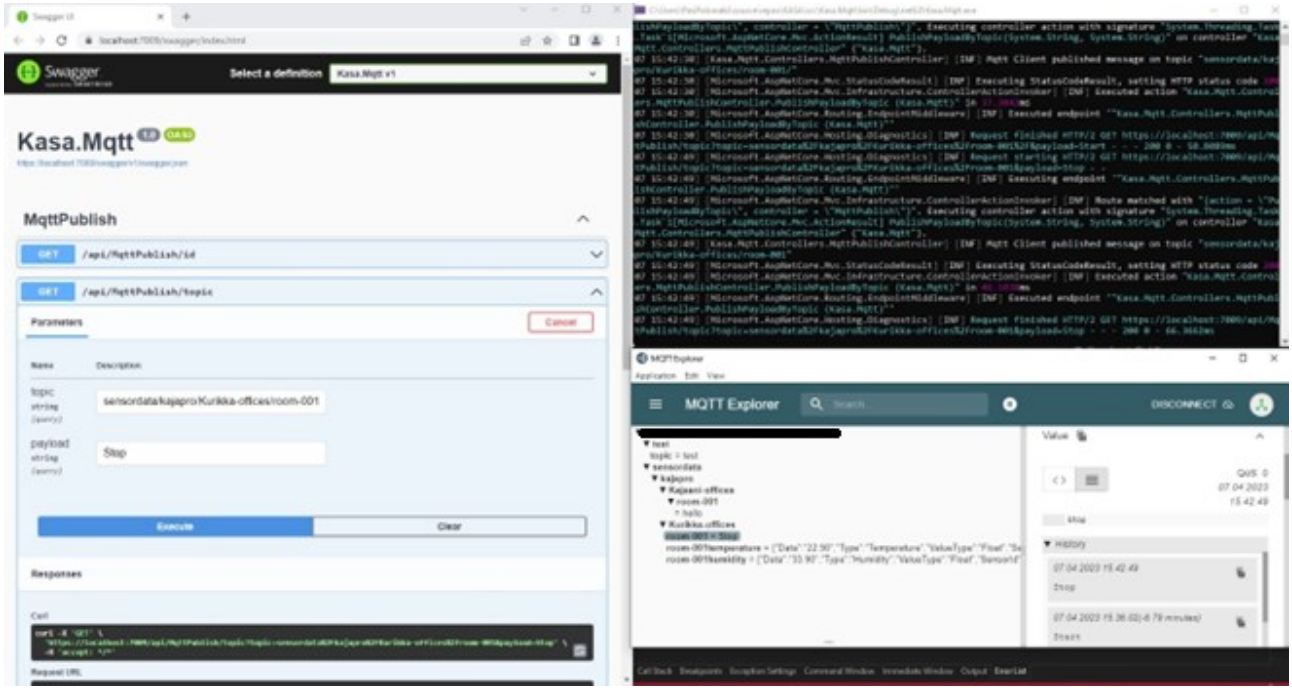
Kuva 13. MQTT-viestit ja laitteen debuggaus sarjaportista.

5.2 Verkkopalvelimen toteutus

Verkkopalvelut on toteutettu osittain Docker-kuvatiedostona ja normaalina palvelinohjelmistona, josta voidaan tarvittaessa tehdä Docker-kuvatiedosto. MQTT-viestipalvelimena toimii Mosquito, jota ajetaan Docker-imagena Azure pilvipalvelussa. Tämän takia tätä palvelua voidaan tarvittaessa ajaa Azuren pilvipalvelussa tai omalla palvelinkoneella. Verkkopalvelimessa on myös viestinhallintapalvelu, käyttöliittymä ja tietokanta. Näissä edellä mainituissa on käytetty samaa toteutusta, jossa voidaan pystyttää nämä palvelut Azuren pilvipalveluun tai omalle palvelinkoneelle. Azure-palveluiden käyttöön liittyy useita kustannuksiin liittyviä ongelmia, jotka voivat tehdä siitä liian kalliin käyttää. Yksi tärkeimmistä ongelmista on hinnoittelun monimutkaisuus. Se sisältää lukuisia muuttujia, jotka voivat vaikeuttaa arviota käyttökustannuksista. Yksi haaste on käyttötapojen arvaamaton luonne, mikä voi johtaa äkillisiin käyttö- ja kustannuspiikkeihin.

Kuvassa 14 näkyy kehitysvaiheessa ja kehitysympäristössä käytetään Swaggeriä, jolloin saadaan helposti testattua API-komennot ja niiden palauttamat vastaukset. Kun ohjelmasta tehdään julkaisukäännöstä, niin asetetaan Debug-lippu pois käytöstä ja kaikki siihen sisältyvät osiot sivuutetaan julkaisuversiota käännettäessä. Tällöin myöskään Swagger ei käänny julkaisuvedokseen. MQTT-explorilla pystytään ottamaan yhteys Mosquito MQTT-palveluun ja tarkkailemaan viestejä paremmin API-kutsuja, jotka käännetään MQTT-

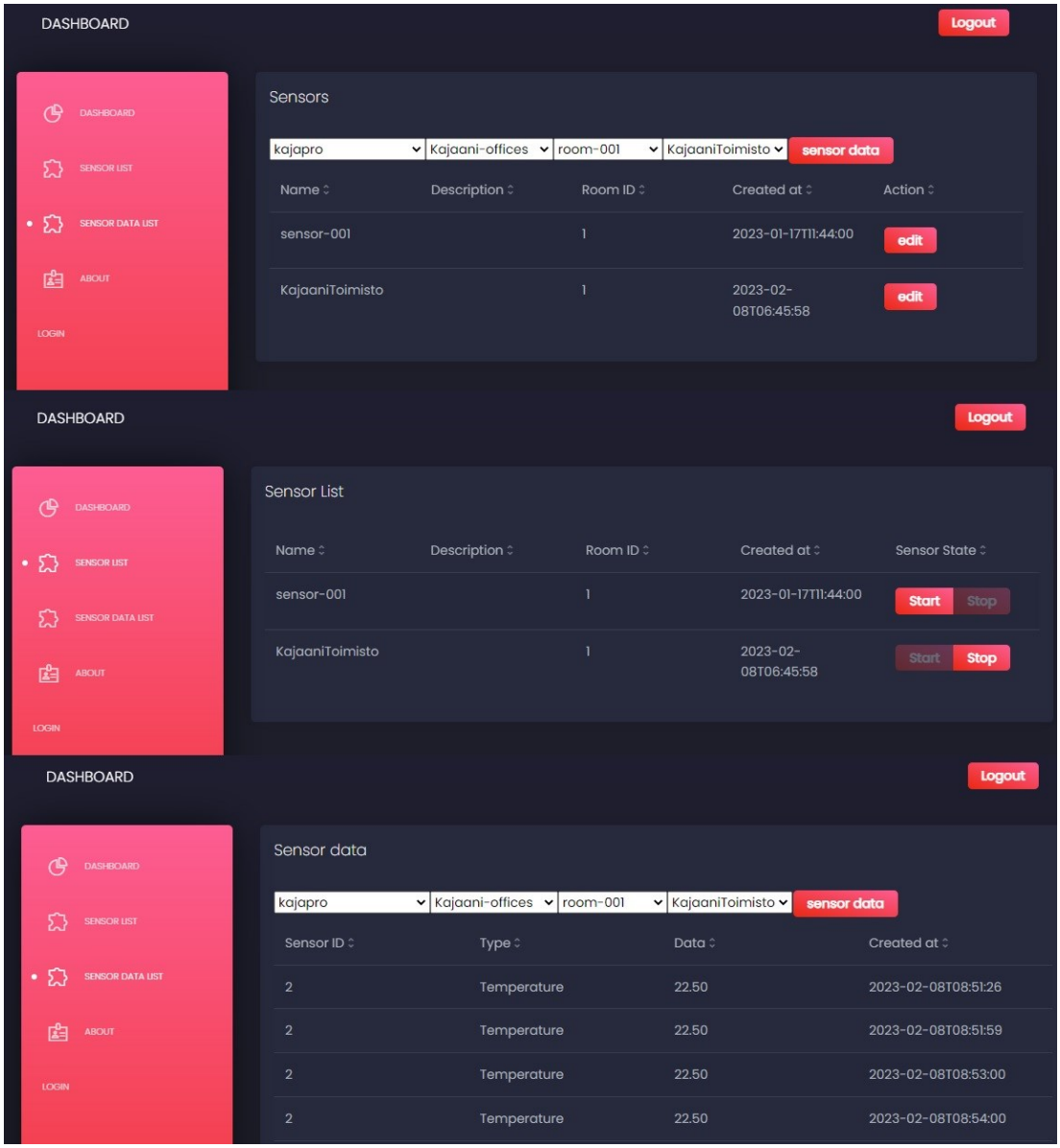
käskyiksi ja lähetetään MQTT-välittäjälle. Nämä palvelimen MQTT-clientin lähettämät viestit on myös salattu käyttäen OpenSSL:ää.



Kuva 14. API-kutsujen ja MQTT-käskyjen debuggaus.

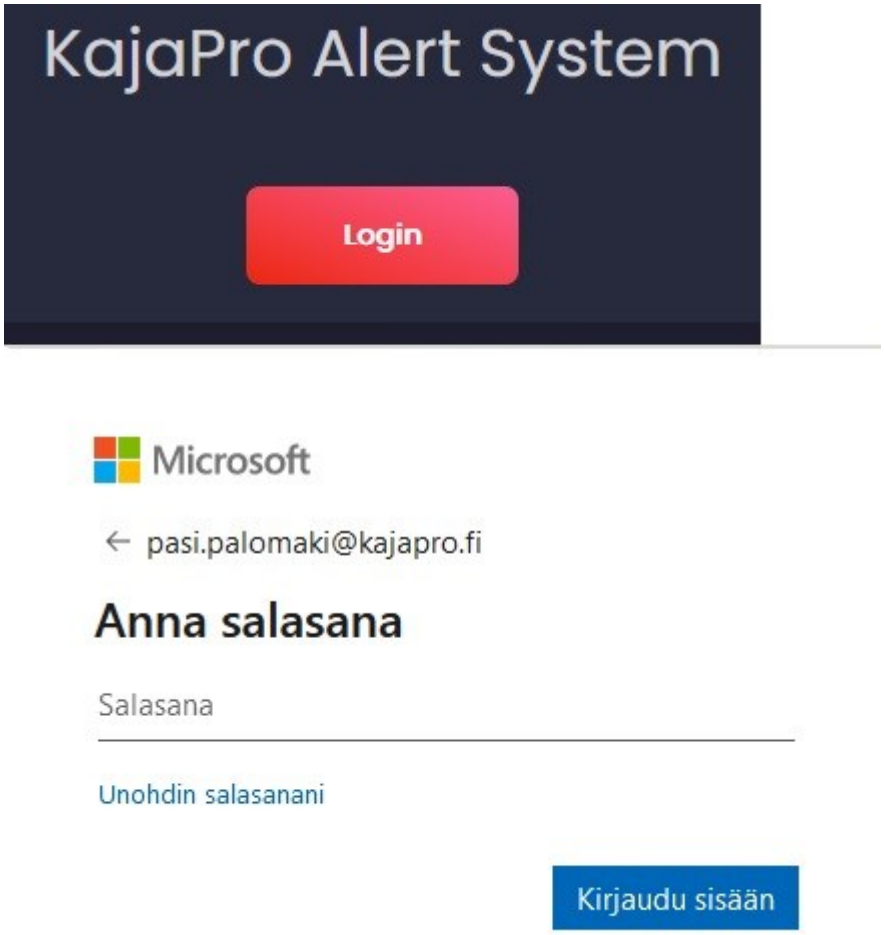
5.3 Käyttöliittymä ja tunnistautuminen

Kuvassa 15 on käyttöliittymän ulkoasu, jota on käytetty käyttäjän käyttöliittymässä ja siinä on käytetty Black-Dashboardin kirjastoa pohjana. Dashboardissa oli osaksi valmista pohjaa datan visualisoimiseen ja käsittelyyn graafista tarkoitusta varten. Alkuun haluttiin käyttöliittymälle näkymään Sensorit, josta voidaan editoida tietyn lokaalien sensoreita. Sensorit-listalle haluttiin mahdollisuus pysäyttää ja käynnistää sensori tarpeen mukaan. Sensori dataan haluttiin, että voidaan seurata tietyn sensorin tuottamaa dataa.



Kuva 15. Käyttöliittymän ulkoasu.

Käyttöliittymä saatiin kuvassa 16 näkyvästi toteutettua Azure AD ja tässä käytetään O365-kirjautumisympäristöä. Käyttöliittymä toimii alkuvaiheessa loppukäyttäjälle nopeana sensori datan lukemisena, joka on yleiskäytännöllisen kirjautumisen takana. Järjestelmään tahdottiin yleisorganisaatiollinen kirjautuminen. Tällöin käyttäjä pystyy käyttämään organisaationsa kirjautumistunnusta ja tunnustenhallinta helpottuu sekä käyttäjän ei tarvitse erikseen luoda kirjautumistunnuksia. Kuitenkin Microsoft-käyttäjätunnuksen vaarantuessa mahdollisella hyökkääjällä voi olla pääsy järjestelmään.

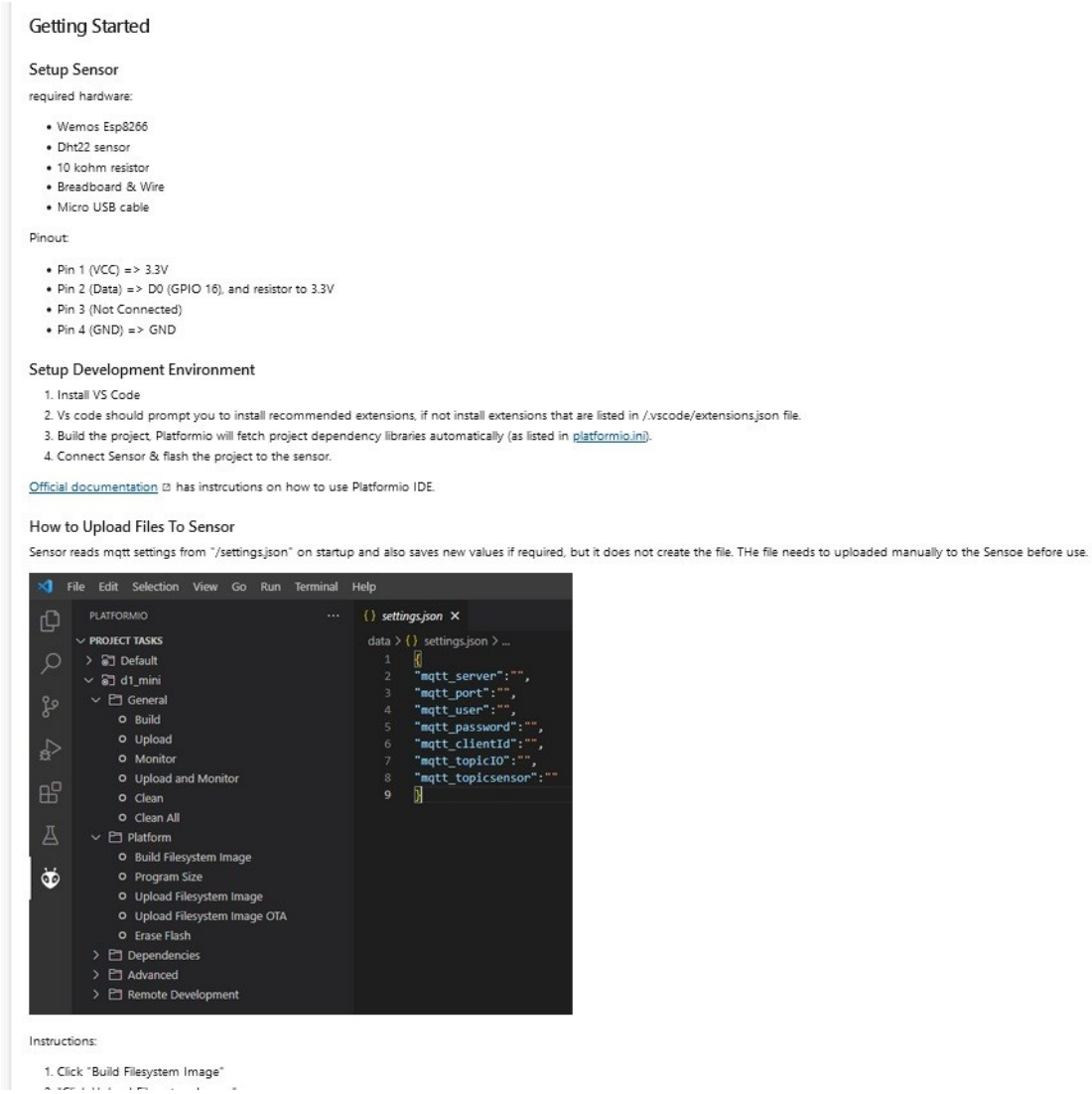


Kuva 16. Käyttöliittymään kirjautuminen.

5.4 Dokumentaatio ja versiohallinta

Microsoftin Azure DevOps valikoitui pilvipalvelu alustaksi tähän toteutukseen, joten oli luonnollista käyttää DevOpssin tarjoamaa versiohallintaa, käännösympäristöä ja testausympäristöä. Versiohallintajärjestelmä oli Git, joka soveltuu eri kehitysympäristöihin ja on helppo käyttää pienen perehtymisen jälkeen. Kaikki ohjelmistokoodit, Docker-asetukset, ohjelmistokoodin dokumentaatio ja arkkitehtuurikuvat säilöttiin Azure DevOpssin Git-ympäristöön. Sekalaisten palveluiden asetus-tekstitiedostot myös säilöttiin Azure DevOpssin Gittiin, jolloin ne pysyvät ajan tasalla ja niistä jää kirjaukset muutoksista. Näin pystytään palauttamaan aikaisempi versio, jos viimeisimpään versioon pääsee ei-kääntyvää tai toimimatonta koodia. DevOpssin Git-hallinnassa kaikki eri tietovarastostot pystyttiin kätevästi jakamaan oman alueensa alle ja hallinta oli helppoa DevOpssin UI:lta.

Dokumentaatiota kirjoitettiin koodiin ja käyttöönotto-ohjeita README tiedostoihin, jolloin ne tulivat kätevästi DevOpssin verkko UI:lla näkymään. Samalla pystyttiin upottamaan kuvia ja tehostamaan tekstiä, jolloin se on selkeämpää lukea. Kuvassa 17 on esimerkkinä sensori tietovarasto aloitusohjeet ja yleiskuvausta käyttöönotosta, jossa on kerrottu sensorista perusasioita ja demovaiheen asetuksia.



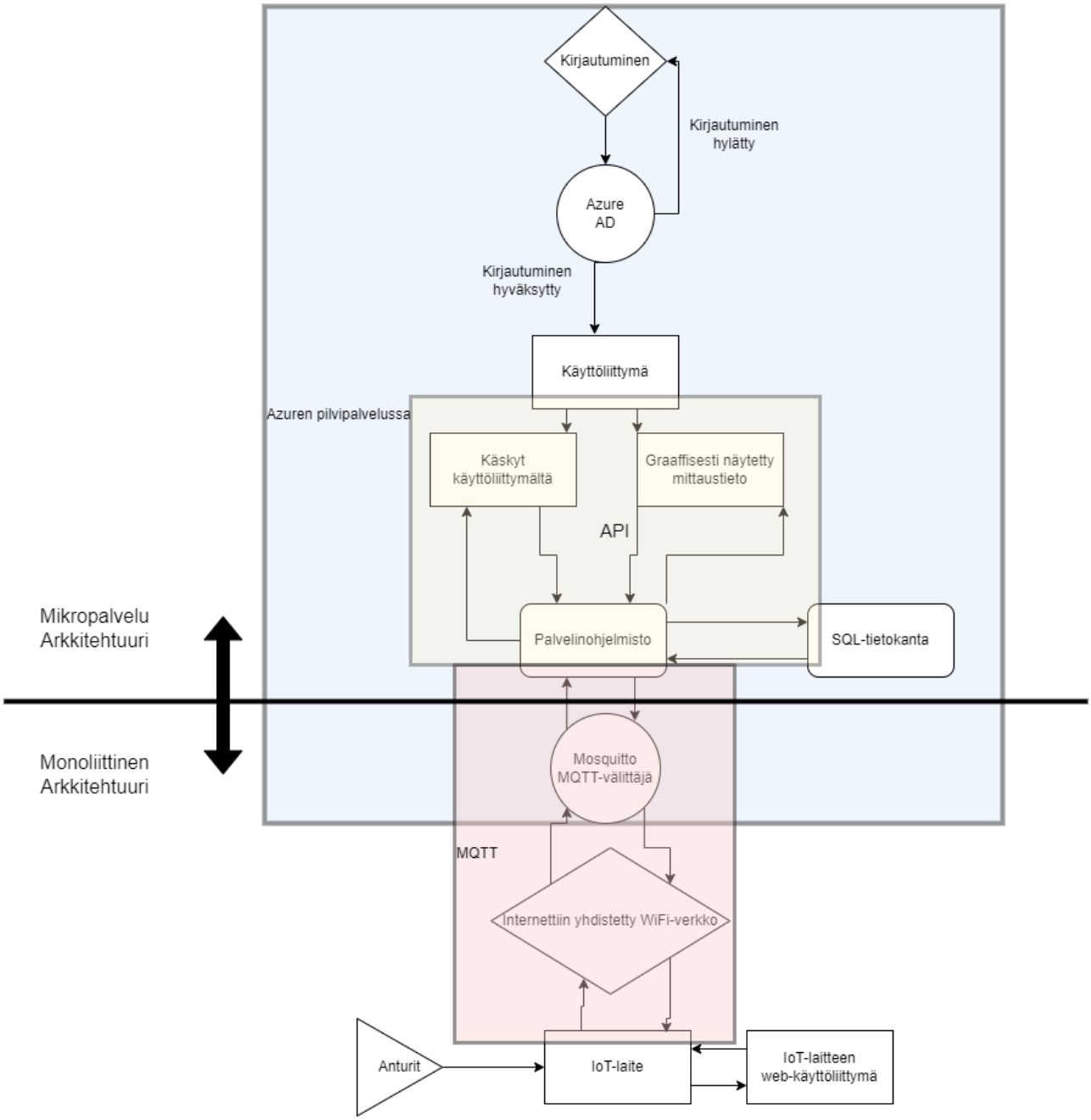
Kuva 17. Esimerkki dokumentin visualisoinnista.

DevOpssissa pystyttiin myös samalla tekemään kehitysputki, jonka läpi tietovarastossa oleva koodi menee. Koodin mennessä putkiston läpi sille tehdään yksikkötestit, integraatiotestit ja lopuksi tulee ulos käännetty applikaatio, joka on valmis laitettavaksi testiympäristöön tai tuotantoon. Tällä pystytään automatisoimaan työtä, joka on aikaisemmin pitänyt tehdä käsin sekä saadaan pidettyä ohjelmiston laatu samana. Myös kehityksessä

tapahtuneet virheet, jotka rikkovat tai eivät läpäise testejä tulevat esiin. Tällöin niihin ehditään puuttua ajoissa, jolloin virheillä ole mahdollisuutta päästä tuotantoon. Ohjelmistokokonaisuuksista pystytään säätämään julkaisu- eli käännösversiot ja samalla määrittelemään Azure DevOps infran päällä oleva käyttöönotto julkaisuversioon heti, kun se on läpäissyt testiputken. Tällä pystytään nopeasti samaan testattua koodia verkkopalveluun tarvittaessa tai ajoitettuna huoltokatkona.

5.5 Tulokset ja havainnot

Kuviossa 18 on IoT-järjestelmän toteutunut arkkitehtuurirakenne, joka saatiin tehtyä kokeilussa.



Kuvio 18. IoT-järjestelmän toteutunut rakenne.

Arkkitehtuurin suunnittelun aloittamista helpotti opinnäytetyön alussa tapahtunut määrittely, jossa oli määritelty käytettävät laitteet ja tekniikat. Tässä vaiheessa saatiin vähän näkemystä järjestelmän eri osista, josta oli taas helpompi lähteä aloittamaan arkkitehtuurin suunnittelua. Tavoitteena oli suunnitella järjestelmä, joka mahdollistaa IoT-laitteen ja järjestelmän välisen viestinnän. Työ alkoi teoreettisen tiedon hankinnalla järjestelmäarkkitehtuurin perusteista ja eri arkkitehtuurivaihtoehdoista. Tämän jälkeen muodostettiin sopiva järjestelmäarkkitehtuuri. Alkuun aloitettu ylhäältä alaspäin -suunnittelutyyli vaihtui

suunnittelun jälkeen itse toteutusvaiheessa alhaalta ylöspäin. Tämä tuntui tapahtuvan luonnostaan, koska alhaalta ylöspäin tuntui luontevammalta ja helpotti pienten testiohjelmistojen kehittämistä. Havaintona tähän voidaan sanoa, että suunnittelu tuntui luontevammalta lähteä ylhäältä alaspäin, koska näin pystyi muodostamaan kokonaiskuvan sujuvammin. Itse ohjelmistokehityksen toteuttaminen oli luontevampaa alhaalta ylöspäin, koska näin pystyi keskittymään pieneen osioon kerralla. Tämä helpotti toteutuksen perusrakennetta ja siten mahdollisti koko arkkitehtuurirakenteen testaamisen. Tämän jälkeen keskityttiin laajentamaan ja täydentämään kokonaisuutta.

Arkkitehtuurillisesti monoliittinen arkkitehtuuri sopi hyvin IoT-laitteen suunnittelulle, koska IoT-laite oli muusta järjestelmästä erillään omassa kokonaisuudessa. IoT-laitteen suunnittelu onnistui suunnitellusti ja IoT-laite toimi halutulla tavalla. Havaintona kehityksessä huomattiin ohjelmistonkääntämisessä laitteelle ilmenevän eriävyyksiä, kun käytettiin Arduino IDE ja PlatformIOta. Arduino ympäristöä käyttäessä MQTT-kirjastossa ilmeni ongelmana MQTT-yhteyden muodostaminen, mikä toimi epästabiilisti katketen tai jättämällä yhdistymättä. Tämä epästabiilisuus katosi, kun samaa kirjastoa käännettiin PlatformIOlla. Kehityksen lopussa vaihdettiin käyttämään Visual Studio Coden lisäosana PlatformIOta.

Mikropalveluarkkitehtuurin avulla voitiin järjestelmä luoda Azure Cloud -alustalle sekä erilliseen virtuaalipalvelimeen. Azure Cloud -alusta osoittautui sopivaksi mikropalveluiden toteuttamiseen, koska Microsoftin ekosysteemi oli muutenkin käytössä organisaatiossa. Mikropalveluarkkitehtuuri osoittautui hyvin soveltuvaksi verkkopalvelin ohjelmistolle. Tällä tavoin pystyttiin jakamaan palvelinohjelmisto pienempiin osiin, mikä taas helpotti testaamista sekä API ja MQTT-viestien välittymisen toteuttamista.

Palvelinohjelmisto oli riippuvainen arkkitehtuurista ja millaisista vaatimuksista sen toteuttamisympäristöön asetettiin, koska ei haluttu jäädä yhden toimittajan loukkuun tai tilanteeseen, missä palvelut estyvät tai ne olisivat kyberhyökkäyksessa estettyinä. Tässä kirjoitettiin ohjelmisto, jossa oli MQTT-viestipalvelu, SQL-tietokantaan tallennuslogiikka ja API-käskyjen käsittely käyttöliittymälle. Tämä ohjelmisto toimi koko järjestelmän liitoskohtana. Tämän takia Docker-imageiksi palveluiden toteuttaminen oli hyvä päätös. Samalla nämä palvelut olivat melko helppoja rajata omiksi pienemmiksi kokonaisuuksiksi sekä järjestelmävalvonnan kannalta helpottaa diagnostiikkaa verkkopalveluiden ja palvelinohjelmiston

osalta. Tässä osiossa pyrittiin soveltamaan mikropalveluarkkitehtuurin periaatteita, vaikka osa palvelinohjelmistosta olikin melko monoliittinen.

Käyttöliittymä loppukäyttäjälle oli kohtuullisen helppo pystyttää ja suojata käyttäen O365-ympäristön Azure ADta. Näin saatiin helposti vältettyä ylimääräisten tunnusten ja kirjautumisjärjestelmän virittäminen. Ainut heikkous tässä on mahdollinen pääsyn estyminen O365-kirjautumispalveluun, mutta tässä kohtaa se tiedostettiin tietoisena riskinä, joka otetaan tässä vaiheessa. Käyttöliittymälle saatiin tarvittavat datat ja etäohjaukset mikrokontrollerille asti, jolloin pystyttiin testaamaan koko järjestelmän läpi menevä viesti- ja komentoketju.

Järjestelmään kuului myös ohjelmistokehityksen testaus ja laadun varmistuksen automatisointi. Näitä pyrittiin toteuttamaan Azuren DevOps-ympäristössä käyttäen yleiskäytännöllisiä ohjelmistoja ja periaatteita kuten Git:tiä versionhallinta järjestelmänä. Testit kirjoitettiin myös itseohjelmakoodin yhteyteen, jolloin niitä voidaan ajaa kehityseditorin yhteydessä eivätkä ne ole sidoksissa DevOpssin-kehitysputkistoihin, mutta niillä on yhteensopivuus.

Modulaarisesti järjestelmässä eriteltiin kaikki omiksi kokonaisuuksiksi, jotka oli kommunikointirajapinnalla yhdistetty toisiinsa. Tällä tavoin uusi pystytään lisäämään järjestelmään uusia palveluita tai ominaisuuksia helpommin, kun niiden tarvitsee toteuttaa vain yhteensopivuus kommunikointirajapinnan kanssa. Itse ohjelmistojen suoritusympäristö oli modulaarinen, koska kaikki pilvipalvelut ja pilvipalveluohjelmistot pystytään tarvittaessa vaihtamaan kilpailijan tuotteisiin tai omaan palvelinympäristöön. IoT-laitteita voi lisätä tai vähentää, kunhan ne tukevat MQTT-protokollaa. Järjestelmässä ainut ei-modulaarinen osa oli Azure AD kirjautuminen, mikä voidaan toteuttaa myös perinteisellä kirjautumisjärjestelmällä, mutta tämä vaatii kirjautumislogiikan ohjelmoinnin ja siinä käytettävien käyttäjätietojen tallentamisen SQL-tietokantaan.

Tietoturvaa pyrittiin sisällyttämään itse tekemiskäytäntöihin ja ymmärtämään, miksi tällainen lähestymistapa on kannattava. Hyökkäystestausta järjestelmää vastaan ei suoritettu, vaan pääpaino oli tietoturvallisissa työskentelytavoissa, salattavissa tiedoissa ja luvattoman pääsyn estämisessä. Kaikki yhteydet salattiin käyttäen OpenSSL-salausta, mikä tarkoittaa, että kaikki viestintä laitteiden välillä on salattua. Koko järjestelmän käyttöliittymään

kirjautuminen edellytti Microsoftin organisaation kirjautumista, joten käyttäjän oli käytettävä KajaPron Microsoft-organisaatiotunnusta kirjautuakseen sisään.

Järjestelmälle tehtiin äkillisiä kaatumistestejä kehityksen aikana, ja siinä IoT-laitetta resetoitiin ja pilvipalvelun eri komponentteja päivitettiin kesken normaalin ajon, kun ajettiin päivitettyä ohjelmistoversiota sisään. Tällä tavoin puoli tahattomasti tehtiin katastrofitestausta, jossa testataan järjestelmän toipuminen, jos jokin osa äkillisesti kaatuu pois käytöstä ja uudelleen käynnistyy. Toteutuksen kehityksen aikana ilmeni muutama ongelma, jotka korjattiin lähettämään virheviestiä, jolloin järjestelmä jäi odottamaan järjestelmänhaltian toimenpiteitä.

6 JOHTOPÄÄTÖKSET JA POHDINTA

Opinnäytetyössä lähdettiin selvittämään tutkimuskysymyksen mukaisesti soveltuvaa arkkitehtuuria IoT-järjestelmän arkkitehtuuriksi. Teoriassa keskityttiin kahteen eri arkkitehtuuriin ja suunnittelutapaan. Alussa lähdettiin monoliittisella arkkitehtuurilla, mutta suunnitteluvaiheessa päädyttiin johtopäätökseen, että tämä ei ole hyvä koko järjestelmän arkkitehtuuriksi. Monoliittinen arkkitehtuuri soveltui IoT-laitteen suunniteluun, koska kyseessä oli yksinkertainen ja muusta järjestelmästä irrallinen osa, joka liitettiin vain kommunikointirajapinnan yli järjestelmään.

Mikropalveluarkkitehtuuri taas soveltui palvelinohjelmistolle, käyttöliittymälle ja tietokannalle sopivaksi, koska näitä yhdisti sama palvelin ja pilviympäristö. Näiden yhdistävänä tekijänä oli vielä API-yhdyskäytävät, jota pitkin rajapinnat kommunikoivat keskenään. Arkkitehtuuri, jossa nämä osat pilkottiin pienempiin osiin, helpotti kehitystä alussa. Pieninä kokonaisuuksina nämä palvelut oli helppo ymmärtää ja testata rajapintojen välinen kommunikointi.

Johtopäätöksenä kokeilussa järjestelmäarkkitehtuuri pitäisi pitää kevyenä, jolloin painopiste pysyy kokeilun tekemisessä ja toteuttamisessa. Toisaalta samalla järjestelmäarkkitehtuurin on tuettava kokeilua. Jos kokeiluvaiheessa arkkitehtuurissa ilmenee ongelmaa, niin ongelmat on helpompi korjata aikaisessa vaiheessa. Tämän takia arkkitehtuurin dokumentaation on kuvattava tarpeeksi selkeästi järjestelmän arkkitehtuuria, että sidosryhmät pystyvät viestimään keskenään. Onnistuneella kommunikaatiolla pystytään välttämään virheellisiä rajapinta ja arkkitehtuuri valintoja.

IoT-sovelluksen arkkitehtuurin toteutus lähtee IoT-laitteesta tai IoT-järjestelmästä eli toteutusta voidaan toteuttamaan ylhäältä alaspäin tai alhaalta ylöspäin. Tässä määrittelee paljon, onko haluttu mittalaite lähettämään dataa verkkoon vai verkossa oleva järjestelmä yhdistää mittalaitteeseen. Arkkitehtuurillista modulaarisuutta pystyy parantamaan pitämällä riippuvuudet eri teknologioihin vähäisenä ja jakamalla järjestelmä mikropalveluarkkitehtuurin mukaisesti pienempiin osiin omiksi palveluikseen tai ohjelmistoapplikaatioksi. Arkkitehtuurillisesti opinnäytetyössä toteutettiin hyvin modulaarinen ja yleiskäytännöllinen

arkkitehtuuri, jossa pilvipuolen kommunikaatio eri osien välillä oli API-yhdyskäytävässä ja mikrokontrollerin sekä pilven kommunikointi tapahtui MQTT-protokollalla.

IoT-laitteessa itsessään rajoittava tekijä on, mitä ominaisuuksia siinä on sekä suunniteltu käyttöympäristö. Tässä tapauksessa oli sisätilat, joista oli pääsy WiFi-verkkoon ja IoT-laite mittasi vain lämpötilaa sekä kosteutta. IoT-laitteen kommunikointi MQTT:lla sanelee tässä toteutuksessa, mitä viestintäprotokollaa käytetään. MQTT-viestiprotokollan vaihtaminen joksikin muuksi aiheuttaisi uudelleenohjelmointia ja monien asioiden uudelleen miettimistä. Laitekerros rakentui hyvin perinteisenä monoliittisena ohjelmistokehitysrakenteella, joka on tyypillistä sulautetuille järjestelmille niiden luonteen vuoksi.

Tietoturvan parantaminen lähti tiedon salaamisesta ja estämällä luvaton pääsy IoT-järjestelmään. Tiedon salaaminen vaikeuttaa onnistuneen hyökkääjän kaapatun tiedon hyödyntämistä, koska hyökkääjä joutuu purkamaan salauksen ennen kuin pääsee lukemaan tietoa. Vaatimalla vahvoja salasanoja tai käyttämällä organisaatiokirjautumisia estetään turhia käyttäjätunnuksia, jossa käyttäjät kierrättävät heikkoja salasanoja. Tällä tavoin käyttäjätunnusten ollessa yksilöllisiä, lokeihin tallentuu tieto siitä, kuka on järjestelmää käyttänyt.

Koodin katselmoinnilla pyrittiin pitämään laadullisesti koodi hyvänä ja turvallisena suorittaa. Samalla seurataan käytettyjä kirjastoja, onko näihin tullut tietoturvavaroituksia tai suosituksia päivittää uudempaan haavoittuvuuksien takia. Tämä seuranta on jatkuvaa, koska haavoittuvuuksia tulee ilmi tietyin väliajoin ja yleensä liian myöhään, jolloin on kiire päivittää tai paikata tietoturva-aukkoja. Tässä opinnäytetyössä pyrittiin tuomaan kehityskäytännöillä tietoturvaa toteutukseen ja tuomaan ne osaksi kokeilun kehitystä. Tämä nähtiin kokeilussa riittäväksi.

Opinnäytetyön tuloksia pystyy hyödyntämään yleisesti IoT-järjestelmän kehittämisessä, koska kokeilu onnistuttiin toteuttamaan hyvin modulaarisesti. Opinnäytetyön eri osa-alueissa käytettyä arkkitehtuuria, tekniikkaa tai palvelun tarjoajaa voi vaihtaa oman tarpeen mukaan. Huomioitavaa on kuitenkin, että yksinkertaisessa mikrokontrollerissa monoliittinen arkkitehtuuri on yleisesti helpoin valinta. Tietoturvan parantamiseen tehdyt asiat ovat myös yleisesti käytännöllisiä sekä parantavat tietoturvaa muissakin IoT-järjestelmissä.

Jatkokehityksen kannalta olisi viisasta katsoa, kuinka esimerkiksi LoRaWAN ja MQTT pystyttäisiin integroimaan rinnakkain. Tällä voitaisiin lisätä myös LoRAWAN IoT-laitteita osaksi järjestelmää, jolloin järjestelmän yleiskäytännöllisyys kasvaisi. Tässä voitaisiin myös miettiä kameran lisäämistä ESP32:seen, jolloin voitaisiin lähettää IoT-laitteella otettuja kuvia palvelimelle. Onko mahdollista toteuttaa tämä suoraan käyttäen MQTT:tä, vai olisiko viisaampaa luoda erillinen toteutus? Samalla olisi suositeltavaa tarkastella, tukeeko kokeilussa saatu arkkitehtuuri uusia kehityssuuntia.

LÄHTEET

Arduino. (2023). *Arduino IDE* (Version 2.2.1). https://www.arduino.cc/en/software

Aroraa, G. K., Kale, L., & Manish, K. (2017). *Building microservices with .NET Core: Transitioning monolithic architecture using microservices with .NET Core.* Packt Publishing.

Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3. p.). Addison-Wesley.

Blank, A. G. (2004). *TCP/IP Foundations.* San Francisco: Sybex.

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley.

Brown, S. (2017). *Software Architecture for Developers.* Leanpub.

Buyya, R., & Dastjerdi, A. V. (2016). *Internet of Things: Principles and Paradigms.* Morgan Kaufmann.

CyberArk. (i.a.). *Azure Active Directory vs. Okta: What's the Difference?* https://www.cyberark.com/resources

Espressif Systems. (i.a.). *ESP32 Datasheet.* https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

Feathers, M. C. (2004). *Working Effectively with Legacy Code.* Prentice Hall.

Git Contributors. (2023). *Git* (Version 2.42.0). https://git-scm.com/

Goralski, W. 2008. *The Illustrated Network: How TCP/IP Works in a Modern Network.* Morgan Kaufmann.

Gorton, I. (2011). *Documenting a Software Architecture.* https://doi.org/10.1007/978-3-642-19176-3_8

Hentrich, C., & Zdun, U. (2012). *Process-driven SOA: Patterns for aligning business and IT.*

Ingeno, J. (2018). *Software Architect's Handbook- Become a successful software architect by implementing effective architecture concepts.* Packt publishing

Jayaprakash, S. (2008). *Technical writing.* Himalaya Pub. House.

Jorgensen, D. (2002). *Developing .NET web services with XML.* Syngress Publishing.

Kanjilal, J. (2013). *ASP.NET Web API: Build RESTful web applications and services on the .NET framework: master ASP.NET Web API using .NET Framework 4.5 and Visual Studio 2013.* Packt Publishing.

Kavis, M. (2014). *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS).* John Wiley & Sons.

Karagiannis, V., Chatzimisios, P., Vazquez-Gallego, F., & Alonso-Zarate, J. (2015). *A Survey on Application Layer Protocols for the Internet of Things* https://www.researchgate.net/publication/303192188_A_survey_on_application_layer_protocols_for_the_Internet_of_Things

Kruchten, P. (1995). Architectural Blueprints—The "4+1" View Model of Software Architecture. *IEEE Software, 12*(6), 42–50.

Lukka, K. (2001). *Konstruktiivinen tutkimusote.* https://metodix.fi/2014/05/19/lukka-konstruktiivinen-tutkimusote/

Martin, R.C. (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hill.

Microsoft. (i.a. -a). *Azure Cloud Services.* https://azure.microsoft.com/en-us/services/

Microsoft. (i.a. -b). *What is Azure Active Directory?* https://azure.microsoft.com/en-us/services/active-directory/

Microsoft. (i.a. -c). *What is platform as a service (PaaS)?* https://azure.microsoft.com/en-us/overview/what-is-paas/

Microsoft Corporation. (2023). *Visual Studio Code* (Version 1.84.0) https://code.visualstudio.com/

Microsoft Corporation. (i.a.). *Azure DevOps.* https://dev.azure.com/

Miller, M. (2015). *The Internet of Things: How Smart TVs, Smart Cars, Smart Homes, and Smart Cities are Changing the World.* Que Publishing.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media.

Ojasalo, K., Moilanen, T., & Ritalahti, J. (2015). *Kehittämistyön menetelmät: Uudenlaista osaamista liiketoimintaan* (3.–4. painos.). Sanoma Pro.

OneLogin. (i.a.). *Identity and Access Management Solutions.* https://www.onelogin.com/

OpenSSL Project. (i.a.). *OpenSSL Wiki.* https://wiki.openssl.org/index.php/Main_Page

Ping Identity. (i.a.). *Identity and Access Management Solutions.* https://www.pingidentity.com/

PlatformIO. (2023). *PlatformIO for Visual Studio Code* (Version 3.3.1). https://platformio.org/

Pressman, R. S. (2009). *Software Engineering: A Practitioner's Approach.* McGraw-Hill Education.

Richardson, C. (2019). *Microservices Patterns.* Manning Publications.

Rozanski, N., & Woods, E. (2011). *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Addison-Wesley.

Srirama, S. N., & Buyya, R. (2019). *Fog and Edge Computing: Principles and Paradigms.* https://doi.org/10.1002/9781119525080

Strimbei, C., Dospinescu, O., Strainu, R-M., & Nistor, A. (2015). Software Architectures-Present and Visions. *Informatica Economică, 19*(4),13–27. https://www.researchgate.net/publication/289528571_Software_Architectures_-_Present_and_Visions/link/568ec24408ae78cc05160880/download

Subramanian, H., & Raj, P. (2019). *Hands-on RESTful API Design Patterns and Best Practices.* Packt publishing.

The IoT Security Foundation. (i.a). https://iotsecurityfoundation.org/best-practice-guidelines/

Thomas, D., & Hunt, A. (1999). *The Pragmatic Programmer: Your Journey to Mastery.* Addison-Wesley.