gRPC:n hyödyntäminen mikropalveluiden synkronisessa kommunikoinnissa

Juuso Hakala

Haaga-Helia ammattikorkeakoulu

Tradenomi

Amk-opinnäytetyö

2024

Tiivistelmä

Tekijä(t)
Juuso Hakala

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
gRPC:n hyödyntäminen mikropalveluiden synkronisessa kommunikoinnissa

Sivu- ja liitesivumäärä
57 + 16

Tämä toiminnallinen opinnäytetyö käsittelee mikropalveluiden synkronista kommunikointia gRPC-nimisen teknologian näkökulmasta. Työssä perehdytään siihen, miten gRPC:stä voidaan hyötyä mikropalveluiden kehityksessä, miten se parantaa mikropalveluiden kommunikointia, ja miten gRPC:tä voidaan käyttää käytännössä. Työ keskittyy mikropalveluiden kommunikointiin, joten koko mikropalveluarkkitehtuuria ei käsitellä.

Työ alkaa teoriaosuudella, jossa esitellään aluksi mikropalveluarkkitehtuuria ja mikropalveluiden piirteitä. Tästä edetään tutkimaan palvelimien rakennetta, toimintaa ja virtualisointia. Lisäksi tutkitaan mikropalveluiden ajamista palvelimilla ja mikropalveluiden ajoympäristön keskeisimpiä kommunikointiin liittyviä komponentteja. Seuraavaksi selvitetään, miten mikropalvelut voidaan integroida yhtenäiseksi kokonaisuudeksi API:en avulla. Mikropalveluiden eri kommunikointitapoja esitellään, jossa pääpainona keskitytään synkroniseen kommunikointiin. Lopuksi perehdytään gRPC:hen, jonka osalta tarkastellaan sen hyötyjä ja haittoja, sekä soveltuvuutta mikropalveluiden synkroniseen kommunikointiin. Hyötyjen kannalta käsitellään esimerkiksi HTTP/2-protokollan ja Protocol Buffers -nimisen datan sarjoittamismuodon käyttöä. Nämä nopeuttavat tiedonsiirtoa mikropalveluiden välillä binäärimuodon ansiosta. Teoriaosuuden lopussa käsitellään gRPC:n eri viestintämalleja, toimintaa ja toteuttamista ohjelmistoihin.

Teoriaosuuden jälkeen alkaa työn toiminnallinen osuus, jossa kuvataan tämän opinnäytetyön osana kehitetyn mikropalveluprojektin toteutus alusta loppuun. Projektin kehitys koostui eri vaiheista, jotka käsitellään johdonmukaisesti toteuttamisen mukaisessa järjestyksessä. Projektin toteutuksessa hyödynnettiin ohjelmistokehityksen menetelmiä ja työkaluja.

Projekti on esimerkkiprojekti, jonka tarkoituksena on täydentää tietoperustan sisältöjä käytännön toteutuksella. Sen avulla voidaan näyttää, miten eri gRPC API:n rakentamisen vaiheet voidaan toteuttaa mikropalveluille, ja miten mikropalvelut voidaan integroida toimivaksi kokonaisuudeksi toteutettujen API:en avulla. Projekti koostuu kolmesta Go-ohjelmointikielellä ohjelmoidusta mikropalvelusta, jotka kommunikoivat keskenään synkronisesti tekijän suunnitteleman kommunikointiketjun mukaisesti. Synkroninen kommunikointi toteutettiin gRPC:llä hyödyntäen Protocol Buffersia datan sarjoittamismuotona ja rajapinnan määritelmäkielenä. Kommunikointiketjussa mikropalvelut lähettävät eri verkkopyyntöjä toisillensa etäproseduurikutsujen avulla tietyssä järjestyksessä. Jokaisen mikropalvelun gRPC API:a voidaan testata erikseen.

Tuloksena saatiin aikaiseksi mikropalveluiden lähdekoodit, gRPC API:en määritelmät prototiedostoissa ja generoidut gRPC-ohjelmakoodit Go-ohjelmointikielelle. Nämä kaikki julkaistiin avoimena lähdekoodina GitHub-palvelussa. Lisäksi projekti sisältää dokumentaation Markdown-muodossa, jossa ohjeistetaan mikropalveluiden ajaminen, konfiguroiminen ja testaaminen.

Asiasanat
gRPC, mikropalvelu, synkroninen kommunikointi, API, ohjelmistoarkkitehtuuri, ohjelmistokehitys

Sisällys

1 Johdanto

Monet yritykset ympäri maailmaa käyttävät nykyään mikropalveluarkkitehtuuria suurissa sovelluksissaan. Mikropalveluiden integroiminen prosessien välisellä kommunikoinnilla on yksi tärkeimmistä asioista mikropalveluarkkitehtuuria käyttävässä sovelluksessa (Indrasiri & Kuruppu 2020, luku 1 alaluku Summary). Datan liikkuminen mikropalvelusta toiseen on elintärkeä ominaisuus niille. Tähän tarvitaan luotettava ja suorituskykyinen ratkaisu, jotta mikropalvelut voivat kommunikoida keskenään mahdollisimman tehokkaasti. Kaikki tekniikat eivät kuitenkaan sovellu kaikkiin käyttötarkoituksiin, koska mikropalvelut voivat kommunikoida eri tavoilla.

Mikropalveluiden kommunikoinnilla tarkoitetaan datan tai viestien välittämistä mikropalveluiden välillä. Tämän avulla mikropalvelut voivat tehdä yhteistyötä jonkin yhteisen tavoitteen saavuttamiseksi. (Tanpure 7.7.2023.) Tässä opinnäytetyössä perehdytään mikropalveluiden synkroniseen kommunikointiin gRPC-nimisen teknologian näkökulmasta. Tavoitteena on selvittää, miten gRPC:tä voidaan käyttää mikropalveluiden synkronisessa kommunikoinnissa ja miten siitä hyödytään. Lisäksi tavoitteena on oppia lisää aiheesta sekä kehittää ammatillisia ohjelmistokehitystaitoja ja -tietoja. gRPC:tä ei ole tarkoitus käydä läpi yksityiskohtaisesti, koska se pitää sisällään valtavan määrän ominaisuuksia. Tästä syystä sitä käsitellään kokonaisvaltaisesti keskittymällä työn rajaukseen. gRPC:tä tarkastellaan myöhemmin tässä työssä luvussa 4.

Lopputuloksena kehitetään kolmesta mikropalvelusta koostuva esimerkkiprojekti, jossa käytetään gRPC:tä kommunikointiprotokollana palveluiden välillä. Jokainen mikropalvelu rakennetaan kuusikulmaista ohjelmistoarkkitehtuuria käyttäen. Projektissa on toiminto, jonka suorittaminen jakautuu näihin kolmeen palveluun. Palvelut kommunikoivat synkronisesti keskenään gRPC:n avulla suunniteltavan kommunikointiketjun mukaisesti suorittaakseen tämän toiminnon. Tuotoksessa tuodaan esille käytännön näkökulmasta gRPC API:n rakentaminen ja gRPC:n synkroninen käyttö mikropalveluiden kommunikoinnissa. gRPC voi toimia synkronisesti ja asynkronisesti, mutta tässä työssä kuitenkin keskitytään gRPC:n synkronisiin ominaisuuksiin. Tuotoksessa hyödynnetään kehittämismenetelminä Git-versionhallinnan aktiivista käyttöä, Kanbania ja 12-tekijän sovellusta.

Valitsin työn aiheen suurimmaksi osaksi oman kiinnostukseni pohjalta. Olen ollut kiinnostunut mikropalveluista ja niiden kehityksestä jo jonkin aikaa, ja minulle oli kohtalaisen selvää tehdä opinnäytetyöni aiheeseen liittyen. Mikropalvelut ovat kuitenkin erittäin laaja aihekokonaisuus, josta riittäisi käsiteltävää useammalle eri opinnäytetyölle. Työ siis tarvitsi rajausta. Päädyin rajaamaan työni mikropalveluiden kommunikointiin, koska se oli minulle kiinnostavimpia aihealueita mikropalveluissa. Toisaalta mikropalveluiden kommunikointi on myös laaja aihealue, joten se vaati lisärajausta. Tässä päädyin keskittymään synkroniseen kommunikointiin ja gRPC:hen. Valitsin gRPC:n, koska se on myös kiinnostanut minua jo jonkin aikaa. Minulla oli siitä työn aloitushetkellä jo

valmiiksi hieman tietämystä ja käytännön kokemusta, joista uskoin olevan apua työtä tehdessä. Lisäksi gRPC on ajankohtainen ja suosittu teknologia mikropalveluiden kehityksessä.

Työllä ei ole toimeksiantajaa, joten kehitettävät mikropalvelut eivät tule oikeaan käyttöön. Vaatimuksena on se, että ne voivat onnistuneesti kommunikoida synkronisesti gRPC:n avulla suunniteltavan kommunikointiketjun mukaisesti. Tämä vaatimus voidaan varmistaa testaamalla jokaista gRPC API:a erikseen lähettämällä niihin pyyntöjä. Mikropalveluiden integraatio voidaan varmistaa kommunikointiketjun avulla testaamalla toteutettavaa toimintoa kokonaisuutena ja tarkastelemalla mikropalveluiden lokitulosteita.

Mikropalvelut kehitetään minimaalisiksi siten, että niistä ilmenee gRPC:n käyttäminen, sekä gRPC-kommunikoinnin toteuttamisen tarvittavat toimenpiteet ja vaiheet. Minimaalisuudella tarkoitetaan sitä, että mikropalveluihin ei kehitetä toiminnallisuuksia, joita ei tässä projektissa tarvita sen toimimiseksi ja onnistumiseksi. Projektissa siis keskitytään gRPC:n toteuttamiseen tekemällä mikropalveluille vain välttämättömät ominaisuudet suunniteltavan kommunikointiketjun testaamiseksi. Mikropalveluita voidaan halutessa myöhemmin jatkokehittää laajemmiksi. Lopputuloksen tarkoituksena on olla hyödyllinen esimerkki- ja oppimisprojekti, jota voidaan hyödyntää esimerkiksi tulevissa projekteissa tai oppimistarkoituksissa. Tästä syystä ohjelmiston lähdekoodi julkaistaan avoimena lähdekoodina, jotta kuka tahansa voisi hyötyä työstä.

Kohderyhmänä on ensisijaisesti ohjelmistokehittäjät, koska työ kohdistuu suurimmaksi osaksi ohjelmistokehitykseen. Ohjelmistokehittäjistä työ on tarkoitettu niille, jotka haluavat tietää ja oppia enemmän gRPC:stä ja mikropalveluista, sekä gRPC:n käytöstä käytännössä. Koska kohderyhmälle halutaan näyttää gRPC:n käyttäminen käytännössä, työn tuotoksella on tässä luvussa aiemmin mainitut vaatimukset, jotta se palvelisi kohderyhmää paremmin.

Työ sisältää paljon teknillisiä asioita, joten lukijalta odotetaan hieman tietämystä tietoteknisistä aihealueista, jotta työ tuottaisi lukijalle mahdollisimman paljon hyötyä. Lisäksi tähän työhön liittyy paljon käsitteitä, jotka ovat olennaisia työn ymmärrettävyyden kannalta. Ne ovat listattuna liitteessä 1.

2 Mikropalveluiden palvelinarkkitehtuuri

Fyysisessä maailmassa rakennetaan rakennuksia yleensä seuraamalla jotakin tiettyä arkkitehtuurityyliä. Jokainen arkkitehtuurityyli pitää sisällään omia valintoja suunnittelusta, ominaisuuksista ja materiaaleista. Sama pätee myös ohjelmistoihin. Sovellukset yleensä käyttävät monien eri arkkitehtuurityylien yhdistelmiä. (Richardson 2018, luku 2.1.2.) Mikropalvelut ovat yksi ohjelmistojen arkkitehtuurityyli, mutta on olemassa myös monia muita tapoja rakentaa sovelluksia. Monet näistä arkkitehtuurityyleistä keksittiin ennen mikropalveluita. Tässä luvussa käydään ensiksi läpi, miten mikropalveluarkkitehtuuriin päädyttiin ja mistä siinä on kyse, jonka jälkeen mikropalveluita tutkitaan tarkemmin.

2.1 Monoliittinen arkkitehtuuri

Perinteinen lähestymistapa sovellusten rakentamiseen on keskittynyt monoliittiseen arkkitehtuuriin. Tässä arkkitehtuurityylissä kaikki sovelluksen palvelut, funktiot ja ominaisuudet ovat lukittu yhdeksi toimivaksi kokonaisuudeksi. (Red Hat 2023.) Kokonaisuus voi olla yksi ajettava tiedosto tai käyttöönotettava komponentti (Richardson 2018, luku 2.1.3). Tämä on hyvin yleinen tapa kehittää sovelluksia. Sitä näkee edelleen vielä paljon pienemmissä ja keskisuurissa sovelluksissa. Esimerkiksi startup-yrityksissä voi olla hyödyllistä lähteä liikkeelle monoliittisella arkkitehtuurilla, koska se on helpoimpia ja halvimpia tapoja kehittää sovelluksia.

Kun monoliittista sovellusta laajennetaan, sen arkkitehtuuri kuitenkin kasvaa monimutkaisemmaksi. Tämä tekee yksittäisten toimintojen optimoimisesta haastavampaa, koska koko sovellus täytyy ottaa huomioon. (Red Hat 2023.) Tämä voi olla haasteellista suurissa organisaatioissa, joissa on paljon kehitystiimejä kehittämässä samaa sovellusta. Isosta monoliittisesta sovelluksesta saattaa myös ajan myötä tulla raskasta ajaa kehittäjän tietokoneella, koska kaikki sovelluksen komponentit ovat lukittu yhteen laatikkoon. Koko sovellus täytyy siis ladata tietokoneen muistiin yhden pienen toiminnon kokeilemiseksi.

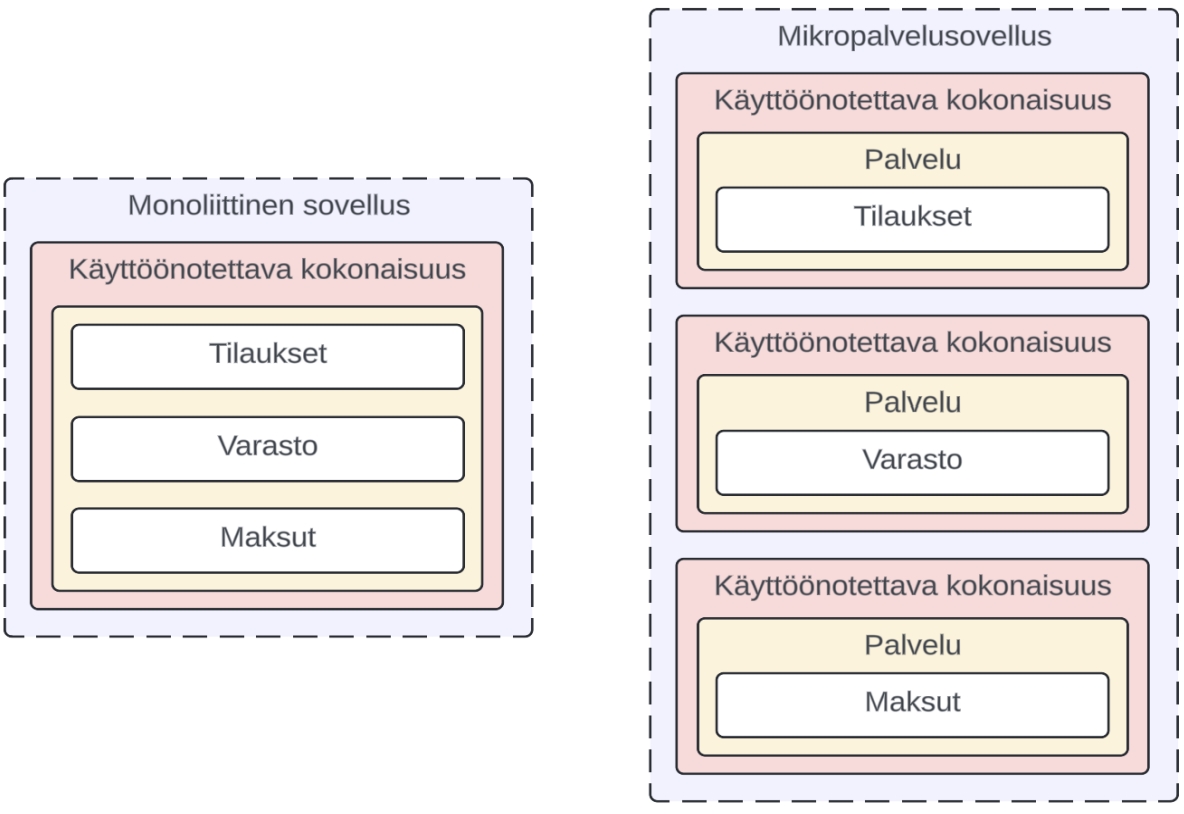
2.2 Palveluihin suuntautunut arkkitehtuuri

Palveluihin suuntautunut arkkitehtuuri on suunnittelutapa, jossa sovelluksen palvelut tekevät yhteistyötä jonkin tavoitteen saavuttamiseksi. Tässä tavassa palvelut ovat tyypillisesti omia prosesseja, joiden kommunikointi tapahtuu verkon yli tehtävillä kutsuilla. (Newman 2015, luku 1 alaluku What about Service-Oriented Architecture.) Tämä siis eroaa huomattavasti monoliittisesta arkkitehtuurista, jossa palveluiden kommunikointi tapahtuu yleensä prosessin sisällä paikallisilla funktiokutsuilla.

Palveluihin suuntautunut arkkitehtuuri syntyi lähestymistapana hoitamaan suuriin monoliittisiin sovelluksiin liittyviä haasteita. Sen tavoitteena on ohjelmistojen uudelleenkäyttö, missä loppukäyttäjien käyttämät sovellukset voivat kaikki käyttää samoja palveluita. Tämä arkkitehtuuri on sisimmissään järkevä idea, mutta sen hyvästä toteutuksesta on ollut paljon yhteisymmärryksen puutetta ohjelmistokehittäjien keskuudessa. Monet tähän liittyvät ongelmat ovat esimerkiksi kommunikointiprotokollan käyttö ja puutteellinen selkeys palveluiden jakamisesta. Palveluihin suuntautunut arkkitehtuuri ei selkeästi määrittele, miten sovellus tulisi jakaa palveluihin ja kuinka jokin iso asia jaetaan pienempään. Tästä muodostui ajan myötä mikropalveluarkkitehtuuri, jossa on otettu huomioon palveluihin suuntautuneen arkkitehtuurin heikkouksia korjaamalla ne. Mikropalveluarkkitehtuuria voidaan ajatella lähestymistapana palveluihin suuntautuneeseen arkkitehtuurin. (Newman 2015, luku 1 alaluku What about Service-Oriented Architecture.)

2.3 Mikropalveluarkkitehtuuri

Mikropalveluarkkitehtuuri on ohjelmiston arkkitehtuurimalli, jossa sovellus jaetaan erillisiin palveluihin. Näitä palveluita kutsutaan mikropalveluiksi, joista jokainen on vastuussa jostakin tietystä sovelluksen logiikan osasta. (Shuiskov 2022, luku 1 alaluku What is a microservice?) Palvelut ovat ikään kuin rakennuspalikoita, jotka tehdään saataviksi muille palveluille, jotta niistä voidaan muodostaa kokonainen järjestelmä (Newman 2021, luku 1 alaluku Microservices at a Glance).



Kuva 1. Monoliittisen sovelluksen ja mikropalvelusovelluksen arkkitehtuurien ero

Kuva 1 havainnollistaa monoliittisen sovelluksen ja mikropalvelusovelluksen arkkitehtuurien eroa (Kuva 1). Mikropalveluarkkitehtuurissa ja palveluihin suuntautuneessa arkkitehtuurissa sovellus jaetaan pienempiin käyttöönotettaviin palveluihin, joiden kanssa on helpompi työskennellä. Mikropalveluarkkitehtuuria voidaan kuitenkin pitää enemmän strategiana sovelluksen kehitystiimeille. (Red Hat 2023.) Jokainen kehitystiimi voi keskittyä tiettyyn sovelluksen alueeseen. Alue voi pitää sisällään yhden tai useamman mikropalvelun, joita kehitystiimi kehittää ja hallinnoi muista kehitystiimeistä irtautuneena. Tämä tekee kehityksestä nopeampaa ja sujuvampaa. (Richardson 2018, luku 1.5.1.) Mikropalveluita käyttämällä ei tarvitse ajaa koko sovellusta yhden toiminnon kokeilemiseksi. Tästä on paljon hyötyä, jos sovellus on erittäin iso, koska se helpottaa sovelluksen hallintaa ja kehitystä.

Mikropalveluarkkitehtuuri on yleensä tarkoitettu suurille yrityksille, joissa on paljon kehitystiimejä kehittämässä samaa sovellusta. Mikropalveluiden käyttäminen voi tulla kalliiksi ja niiden rakentaminen voi olla haastavaa monimutkaisemman arkkitehtuurin takia. Tästä syystä mikropalveluarkkitehtuuri ei ole kovin kannattava pienille startup-yrityksille. Mikropalveluarkkitehtuuriin voidaan kuitenkin myöhemmin siirtyä, jos liiketoiminta ja kehitystiimit kasvavat riittävän suuriksi, jolloin mikropalveluarkkitehtuuri alkaa olla järkevä. (Xu & Lam 11.10.2022, 3:30-4:30 min.) Mikropalveluarkkitehtuuria ei kuitenkaan kannata pitää tavoitteena, johon täytyisi joku päivä siirtyä. Sen käyttäminen riippuu täysin organisaatiosta ja liiketoiminnasta. Monesti monoliittisella arkkitehtuurilla voidaan selvitä hyvin pitkälle.

Mikropalveluarkkitehtuuria käyttävillä yrityksillä on käytössä liiketoiminnassa vaihteleva määrä mikropalveluita. Määrä voi vaihdella kymmenistä satoihin ja sadoista yli tuhanteen. (Shuiskov 2022, luku 1 alaluku What is a Microservice?) Mikropalvelut tuovat näille yrityksille paljon hyötyjä, mutta samalla myös haasteita. Yksi näistä haasteista on mikropalveluiden välinen kommunikointi. Koska mikropalvelut ovat hajautettuja ja irti toisistaan, toimintojen suorittaminen mikropalveluista koostuvassa sovelluksessa riippuu sen palveluiden onnistuneesta kommunikoinnista. (Broshar 23.3.2021.)

2.4 Mikropalveluiden piirteet

Newman kirjoittaa kirjassaan Building Microservices, 2nd Edition, että mikropalvelut ovat luonteeltaan itsenäisiä ja usein liiketoiminta-alueisiin keskittyneitä (Newman 2021, luku 1 alaluku Microservices at a Glance). Hän kuvailee itsenäisyyden ajatuksella sitä, että mikropalveluihin tehdyt muutokset voidaan ottaa käyttöön koskematta muihin mikropalveluihin. Palveluiden itsenäisyys vaatii sen, että ne ovat irti toisistaan. Tämän avulla palveluita voidaan muuttaa ilman, että muihin palveluihin tarvitsee tehdä muutoksia. (Newman 2021, luku 1 alaluku Independent Deployability.)

Liiketoiminta-alueisiin keskittymisellä Newman tarkoittaa sitä, että jokainen mikropalvelu palvelee jotakin tiettyä sovelluksen liiketoiminta-aluetta, jolloin mikropalveluita on helpompi hallita ja niihin on helpompi tehdä uusia toiminnallisuuksia (Newman 2021, luku 1 alaluku Modeled Around a Business Domain). Esimerkiksi mikropalveluarkkitehtuuria käyttävässä verkkokauppasovelluksessa yksi mikropalvelu voi hallinnoida tuotevarastoa ja toinen taas tilauksia. API-tuotteita tarjoava ohjelmistoyritys Kong määrittelee tämänkaltaiset mikropalvelut sovelluksen ydinmikropalveluina (Kong 27.6.2022).

Mikropalvelut tuovat joustavuutta ohjelmistokehitykseen, koska jokainen palvelu voidaan toteuttaa eri teknologioilla. Kehittäjät voivat valita sopivimman ohjelmointikielen ja viitekehyksen jonkin tietyn mikropalvelun toteuttamiseen. Monoliittisessa sovelluksessa kaikki komponentit täytyy usein kehittää samoilla teknologioilla. Tämä rajoittaa uusien tekniikoiden kokeilua tulevaisuudessa. (Richardson 2018, luku 1.5.1.) Mikropalvelut siis antavat mahdollisuuden valita oikean työkalun oikeaan tehtävään. Kaikki ohjelmointikielet eivät odotetusti sovellu kaikkeen ja kaikki eivät myöskään tue kaikkia tarvittavia työkaluja. Tämä teknologiavalinnan tuoma joustavuus parantaa kehittäjien tuottavuutta ja mahdollistaa innovaation nopeuden eri osaamisalueilla (Kong 27.6.2022).

Mikropalveluihin pohjautuva sovellus on hajautettu järjestelmä, joka koostuu useista tietokoneprosesseista, joita ajetaan yleensä useilla eri palvelinkoneilla. Jokainen mikropalvelu on yleensä oma ajettava prosessi. (Microsoft 2022.) Nämä prosessit pitää jollakin tapaa yhdistää toisiinsa, jotta ne voivat kommunikoida keskenään mahdollistaen mikropalveluihin jaetun sovelluksen toimivuuden (Indrasiri & Kuruppu 2020, luku 1). Esimerkiksi tilauksia hallinnoivan mikropalvelun täytyy kommunikoida tuotevarastoa hallinnoivan palvelun kanssa, jotta se voi suorittaa jonkin toiminnon. Se voi lähettää pyynnön tuotevarastopalvelulle, joka vastaa pyyntöön esimerkiksi palauttamalla tuotteiden määrän varastossa. Ilman mikropalveluiden kommunikointia data ei liikkuisi mikropalvelusovelluksen sisällä, eikä useampaan mikropalveluun jakautuvia toimintoja saisi suoritettua. Tästä syystä prosessien välinen kommunikointi on yksi tärkeimmistä asioista mikropalvelusovelluksissa (Indrasiri & Kuruppu 2020, luku 1).

Perinteisessä monoliittisessa sovelluksessa, joka ajetaan yleensä yksittäisessä prosessissa, voidaan toteuttaa sovelluksen eri komponenttien välinen kommunikointi helposti metodi- tai funktiokutsuilla (Microsoft 2022). Mikropalveluissa se ei kuitenkaan ole niin yksinkertaista. Mikropalvelut täytyy yhdistää toisiinsa verkon kautta prosessien välisellä kommunikoinnilla, mikä mahdollistaa tiedon siirtämisen toisesta prosessista toiseen (Indrasiri & Kuruppu 2020, luku 1). Verkon kautta tehtävät kutsut ovat haasteellisia, koska ne lisäävät viivettä sekä virheiden mahdollisuutta (Borysov & Gardiner 3.9.2021).

2.5 Palvelimen merkitys mikropalveluille

Kaikkien ohjelmistojen, mukaan lukien mikropalveluiden, ajamiseen tarvitsee aina fyysisen tietokoneen. Kehityksessä käytetään yleensä omaa tietokonetta, mutta käyttöönotossa kuitenkin halutaan käyttää erillisiä palvelimia. Palvelimella voidaan viitata sekä fyysiseen palvelintietokoneeseen, että palvelinkoneella ajettavaan palvelinohjelmistoon.

Palvelintietokone on tietokone, johon asiakastietokoneet voivat yhdistää internetin tai paikallisen tietoverkon yli. Palvelin tarjoaa palveluita, joihin asiakkaat voivat päästä käsiksi. Palvelut ovat palvelinohjelmistoja, joita palvelintietokone ajaa. Palvelimet eivät kuitenkaan ole tavallisia tietokoneita, joihin yhdistetään. Palvelimella on yleensä oma rooli, mihin sitä käytetään. (PowerCert Animated Videos 5.2.2020, 0:00-2:00 min.) Se voi toimia esimerkiksi tietokantojen ajamisessa tai sovelluksen käyttöönottoympäristönä.

Mikropalvelut ovat erillisiä palvelinohjelmistoja. Mikropalvelut voivat kuitenkin toimia myös palvelinohjelmiston lisäksi asiakasohjelmistona. Tämä tarkoittaa sitä, että ne voivat yhdistää toisiin palvelimiin toisten mikropalveluiden käyttämiseksi. Tietokonetta, joka yhdistää palvelinkoneelle, kutsutaan asiakaskoneeksi (Monteclaro 7.11.2023). Asiakasohjelmisto taas on asiakaskoneella oleva ohjelmisto, joka käyttää palvelinkoneella olevaa palvelinohjelmistoa.

Palvelinten täytyy olla koko ajan saatavilla, koska ne ovat elintärkeitä organisaatioille. Vikatilanteiden sattuessa organisaatio tai liiketoiminta voisi vaarantua. Vaikka tavalliset työpöytäasemat voivatkin toimia palvelimina, niitä ei ole tarkoitettu käsittelemään suurta määrää samanaikaisia yhteyksiä. Tämän takia tarvitaan erillisiä palvelintietokoneita, jotka voivat käsitellä samanaikaisesti suuren määrän yhteyksiä tehokkaasti ja luotettavasti. (PowerCert Animated Videos 5.2.2020, 1:15-3:00 min.)

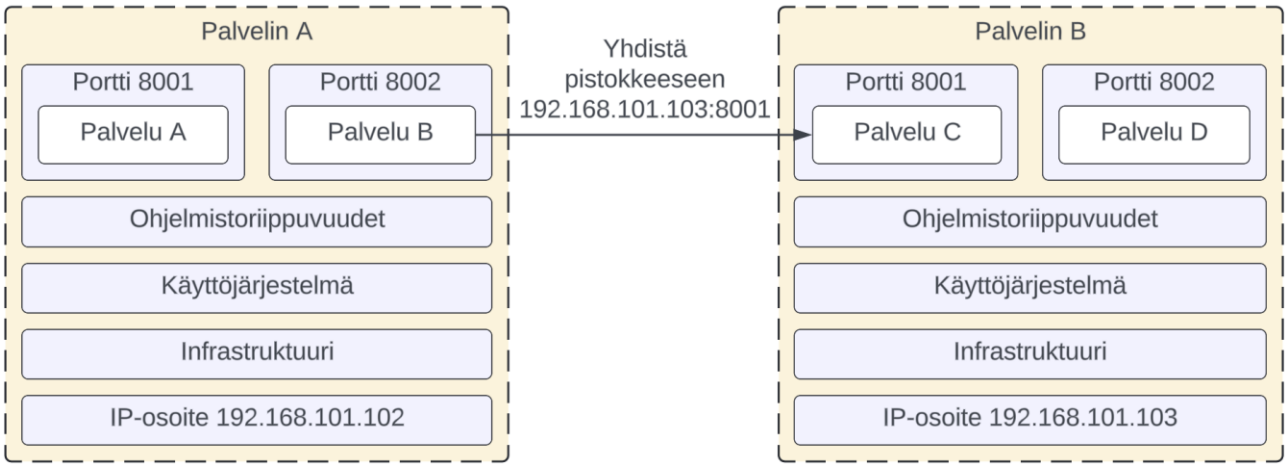
Palvelintietokone tarvitsee palvelinkäyttöjärjestelmän, jonka päällä palvelimen tarjoamat palvelut ajetaan. Palvelinkäyttöjärjestelmät eroavat työpöytäkäyttöjärjestelmistä siinä, että ne ovat vakaampia palvelinten käyttötarkoituksiin, kuten tuhansien samanaikaisten yhteyksien käsittelyyn. (PowerCert Animated Videos 5.2.2020, 5:20-5:45 min.) Windows Server ja Linux-ydintä käyttävät jakelut ovat yleisiä palvelinkäyttöjärjestelmiä. Linux tarjoaa joustavuutta ja loistavan tietoturvan, kun taas Windows Server on enemmän käyttäjäystävällisempi. (Monteclaro 7.11.2023.)

Koska palvelinten täytyy käsitellä hyvin paljon verkkoliikennettä, ne tarvitsevat paljon suorituskykyä. Tämän takia palvelinkoneille tulisi valita mahdollisimman paljon prosessoritehoa. Keskusmuistia tarvitsee palvelinohjelmistojen ajamiseen. Mikropalveluita ajavat palvelimet tarvitsevat paljon keskusmuistia, jos palveluita ajetaan paljon samalla palvelimella. Mikropalveluita ajavat palvelimet

eivät yleensä tarvitse suurta määrää tallennustilaa, koska datan tallentamiseen kannattaa käyttää erillisiä tietokantapalvelimia, jotka ovat suunniteltu datan varastoimiseen.

Jokaisella palvelinkoneella on oma IP-osoite, joka yksilöi palvelimen tietoverkossa. Palvelinohjelmistot pyörivät palvelinkoneella yleensä porteissa. Tätä IP-osoitteen ja portin yhdistelmää kutsutaan pistokkeeksi (englanniksi socket). Pistokkeen avulla asiakkaat voivat yhdistää palvelimelle ja käyttää palvelua, jota ajetaan pistokkeen määrittelemässä portissa. Palvelin odottaa ja kuuntelee pistokkeessa palvelimelle tehtäviä pyyntöjä asiakkailta. Data kuljetetaan kuljetusprotokollan, kuten TCP:n, avulla pistokkeiden välillä. (Oracle 2022.)

Pistokkeet mahdollistavat asiakkaan ja palvelimen välisen tiedonsiirron, mikä on erittäin tärkeätä mikropalveluissa niiden kommunikoinnin perustamiseksi. On hyvä huomata, että IP-osoitteisiin pohjautuvien pistokkeiden lisäksi on olemassa myös muita pistoketyyppejä, kuten Unix-pistokkeet. Mikropalveluiden kommunikoinnissa, joka tapahtuu verkon yli usein eri palvelinkoneiden välillä, on kuitenkin yleistä käyttää IP-osoitteisiin pohjautuvia pistokkeita. Kuvasta 2 nähdään palvelinrakenne, jossa palvelimella A oleva palvelinohjelmisto yhdistää palvelimella B olevaan palvelinohjelmistoon. Yksi mikropalvelu voi kuitenkin käyttää montaa eri porttia. Esimerkiksi gRPC API ja REST API ajetaan eri porteissa, koska ne kuuntelevat erityyppisiä pyyntöjä.



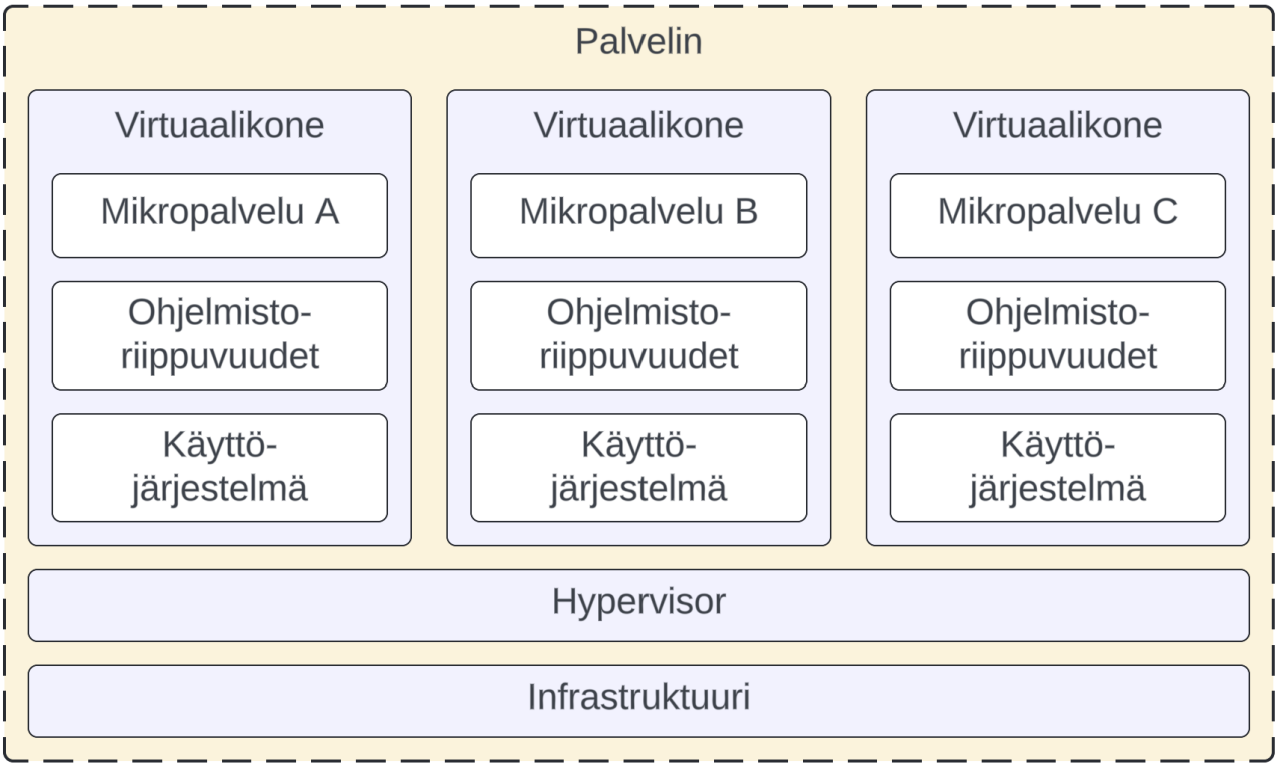
Kuva 2. Palvelinrakenne, jossa palvelinohjelmisto yhdistää toisella palvelimella olevaan palvelinohjelmistoon

2.6 Virtualisoinnin käyttö palvelimilla

Mikropalveluiden ymmärtämiseksi on hyvä tietää, miten niitä ajetaan palvelimilla. Modernissa maailmassa on harvinaista, että mikropalveluita ajetaan suoraan fyysisillä palvelinkoneilla. Nykyään hyödynnetäänkin tietokoneiden ja ajoympäristöjen virtualisointia, mikä on paljon parempi vaihtoehto ohjelmistojen ajamiseen.

2.6.1 Virtuaalikoneet

Varhaisin virtualisointiteknologia, jota vielä tänäkin päivänä käytetään hyvin paljon, on virtuaalikone. Virtuaalikone on ohjelmisto, joka ajaa tietokoneohjelmia ja sovelluksia eristyksessä fyysisestä koneesta. Jokaisella virtuaalikoneella on oma käyttöjärjestelmä, ja jokainen virtuaalikone toimii erillään muista virtuaalikoneista, vaikka ne olisivat kaikki samalla fyysisellä koneella. Virtuaalikoneet voivat jakaa fyysisen koneen resursseja, kuten muistia, prosessoria ja verkon kaistanleveyttä. (Shaw 3.5.2024.) Kuva 3 havainnollistaa mikropalveluiden ajamista virtuaalikoneissa palvelimella.



Kuva 3. Mikropalveluiden ajaminen virtuaalikoneissa palvelimella (mukaillen Buchanan 2024)

Virtuaalikoneita hallinnoi hypervisor, jota voidaan ajaa suoraan fyysisellä palvelinkoneella tai käyttöjärjestelmän päällä. Suoraan fyysisellä palvelinkoneella ajettavaa hypervisoria kutsutaan tyypin yksi hypervisoriksi ja käyttöjärjestelmän päällä ajettavaa tyypin kaksi hypervisoriksi. Tyypin yksi hypervisoria käytetään yleensä palvelinkoneilla, koska sitä pidetään tehokkaampana ja suorituskykyisempänä. (Shaw 3.5.2024.)

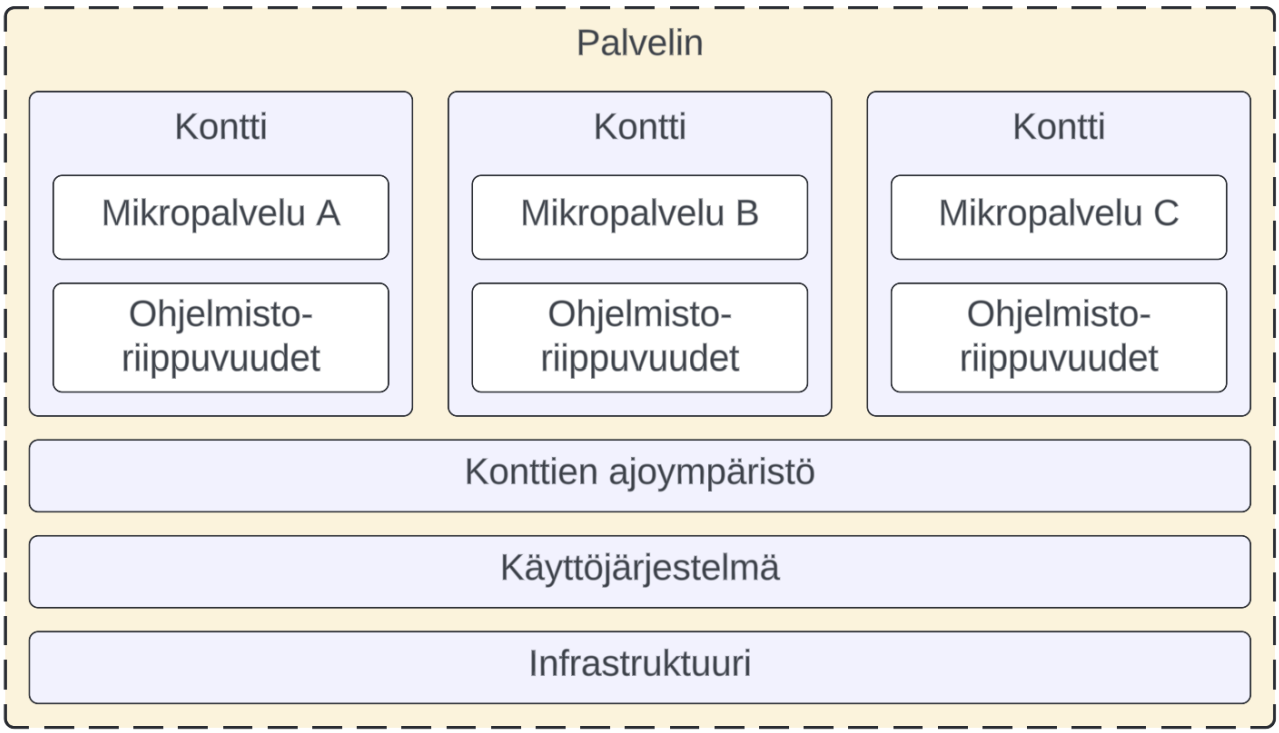
Virtuaalikone ei sinänsä ole mikään uusi asia, koska sen juuret ulottuvat jo 1960-luvulle. Niiden suosio on kuitenkin kasvanut viimeisen kahdenkymmenen vuoden aikana, kun yritykset alkoivat virtualisoimaan palvelimia. Tavoitteena oli vähentää fyysisten palvelinten määrää säästämällä tilaa datakeskuksissa. (Shaw 3.5.2024.) Nykyään ei ole järkeä ostaa esimerkiksi kahta fyysistä

palvelintietokonetta, jos yksi palvelin voi ajaa kahta virtuaalikonetta. Tällä säästetään tilan lisäksi kustannuksia.

Virtuaalikoneet käyttävät paljon tietokoneen resursseja, koska jokainen virtuaalikone luo virtuaalisen kopion palvelimesta. Tämä virtuaalinen kopio sisältää käyttöjärjestelmän ja laitteiston emuloinnin. Lisäksi jokaisesta virtuaalikoneen käyttöjärjestelmästä saattaa joutua maksamaan lisenssin, mikä voi tulla kalliiksi. Nykyään onkin olemassa kevyempi, nopeampi ja kustannustehokkaampi tapa ohjelmistojen virtualisointiin, jossa ohjelmistoja ajetaan konteissa. (Hillpot 12.7.2023.)

2.6.2 Kontit

Konttien avulla voidaan virtualisoida useampia ohjelmistojen ajoympäristöjä samalla käyttöjärjestelmän ytimellä. Kontit sisältävät ainoastaan tarpeelliset asiat ohjelmiston ajamiseksi. Tämä on suosittu tapa ajaa mikropalveluita, koska siitä on enemmän hyötyä kuin jos palveluita ajettaisiin omissa virtuaalikoneissa tai samalla palvelimella. Kontti sisältää oman tiedostojärjestelmän, keskusmuistin, prosessorin ja kaikki mikropalvelun tarvitsemat tiedostot paketoituna yhdeksi ajettavaksi kokonaisuudeksi. (Hillpot 12.7.2023.)



Kuva 4. Mikropalveluiden ajaminen konteissa palvelimella (mukaillen Buchanan 2024)

Jokainen mikropalvelu halutaan yleensä ajaa omassa kontissa. Kahden palvelun ajaminen samassa kontissa ei yleensä ole järkevää, sillä se veisi mennessään konttien tuomia etuja, kuten resurssien eristämisen. Mikropalveluiden ajaminen konteissa tarkoittaa sitä, että jokainen

mikropalvelu voi toimia itsenäisesti, mikä on yksi mikropalveluiden päätarkoituksista (Hillpot 12.7.2023). Kuva 4 havainnollistaa mikropalveluiden ajamista konteissa. Konttien ajoympäristö on ohjelmistoteknologia, jonka avulla voidaan ajaa ja hallinnoida kontteja. (Kuva 4.)

Toisin kuin virtuaalikoneet, kontit eivät tarvitse erillistä käyttöjärjestelmän levykuvaa. Tämän ansiosta kontit ovat kevyempiä ajaa, koska niiden luonnin yhteydessä ei tarvitse käynnistää erillistä käyttöjärjestelmää. Konttien ansiosta mikropalveluita voidaan ajaa tehokkaammin säästäen huomattavasti palvelinten resursseja. Tämä vähentää mikropalveluiden ylläpitämisen kustannuksia. (Hillpot 12.7.2023.)

Kontit tuovat paljon hyötyä, mutta niiden hallinnoiminen ei ole aina helppoa. Niiden käytössä tulee tietää paljon esimerkiksi käyttöjärjestelmien toiminnasta ja tietoverkoista. Kun mikropalvelut ajetaan konteissa, niiden välisen kommunikoinnin toteuttamisesta saattaa tulla asteen haastavampi, kuin jos palvelut ajettaisiin ilman kontteja suoraan samalla koneella. Konttien hallintaan on kuitenkin nykyään olemassa paljon teknologioita, joista suosituimpia ovat Docker ja Kubernetes (Hillpot 12.7.2023).

Kontteja ei kuitenkaan välttämättä ajeta suoraan fyysisellä palvelimella. Monesti halutaan luoda virtuaalikoneita, joiden sisällä ajetaan kontteja, koska se vähentää fyysisten palvelinten määrää. Nykyään siis käytetään virtuaalikoneiden ja konttien yhdistelmää. On hyvä myös huomioida, että näitä ei aina ajeta yrityksen omistamilla fyysisillä palvelimilla. Kaikilla yrityksillä ei välttämättä edes ole omia fyysisiä palvelimia, koska nykyään ollaan hyvin riippuvaisia pilvipalveluista. Pilvipalveluissa joku muu voi hoitaa palvelimia yrityksen puolesta, mikä helpottaa ja nopeuttaa kehittäjien työtä.

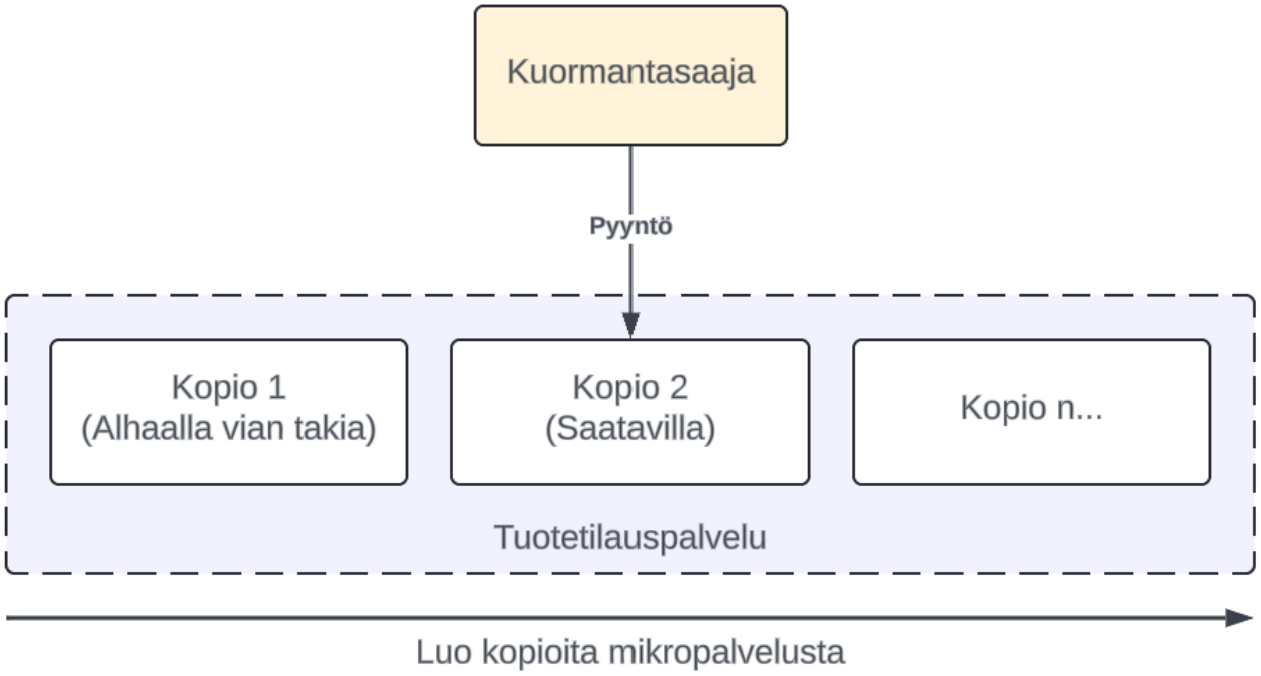
2.7 Mikropalveluiden ajoympäristö

Mikropalveluille on kehitetty omia ajoympäristöjä, jotka eroavat huomattavasti perinteisimmistä ohjelmistojen ajoympäristöistä. Nämä ympäristöt pohjautuvat hyvinkin pitkälle virtualisointitekniikoihin, kuten virtuaalikoneisiin ja kontteihin. Esimerkiksi Kubernetes on nykyään erittäin suosittu ajoympäristö tuotannollisille mikropalvelusovelluksille. Näissä ympäristöissä on pelkkien mikropalveluiden lisäksi paljon enemmän liikkua osia ja lisäkomponentteja, joita kaikissa ympäristöissä ei aina käytetä. Esimerkkeinä voidaan pitää palvelukopioita, kuormantasaajia, palvelun löytämistä ja API-yhdyskäytäviä. Monet komponentit tarvitsevat omia palvelimia, ja kaikki pitää olla integroituna yhdeksi toimivaksi kokonaisuudeksi. Tässä luvussa tarkastellaan muutamia näistä asioista, jotka ovat olennaisia mikropalveluiden kommunikoinnissa.

2.7.1 Palvelukopiot

Yksi mikropalveluiden keskeisistä hyödyistä on yksittäisten palveluiden skaalaus. Tämä tarkoittaa sitä, että yksittäisestä mikropalvelusta voidaan tehdä useita kopioita vastaamaan sovelluksen nykyistä kuormitusta. (Kong 10.3.2022.) Kopioita voidaan kutsua myös instansseiksi. Perinteisissä monoliittisissa sovelluksissa skaalaus täytyy tehdä kaikille sovelluksen komponenteille, vaikka vain yhdelle sovelluksen komponentille tulisi enemmän kuormitusta kuin muille. Mikropalveluissa tämä ei ole ongelmallista, koska jokainen palvelu voidaan skaalata erikseen dynaamisesti. Tämän avulla paljon kuormitusta saavasta palvelusta voidaan tehdä enemmän kopioita ilman, että muita palveluita tarvitsee skaalata samalla. Tämä vähentää kustannuksia säästämällä käytettäviä resursseja ja parantaa sovelluksen suorituskykyä. (Kong 27.6.2022.)

Richardson kutsuu kirjassaan Microservices Patterns tämän kaltaista menetelmää, jossa tehdään identtisiä kopioita sovelluksesta, skaalaukseksi x-akselilla. Kopioita ajetaan kuormantasaajan takana, joka jakaa sovellukseen tulevat pyynnöt kopioiden kesken käyttäen jotakin tiettyä algoritmia. Tämä parantaa sovelluksen kapasiteettia ja saatavuutta. (Richardson 2018, luku 1.4.1.) Mikropalvelusovelluksessa tehtävät kopiot ovat siis kopioita yksittäisistä palveluista, joihin voidaan jakaa verkkopyyntöjä sovelluksen kuormituksen perusteella. Kuvassa 5 on visuaalisesti havainnollistettuna tämä prosessi.



Kuva 5. Mikropalvelun skaalaus, jossa palvelusta on tehty identtisiä kopioita (mukaillen Richardson 2018, luku 1.4.1)

Monesti saatetaan luulla, että jokainen mikropalvelu ajetaan tuotantoympäristössä sellaisenaan. Todellisuudessa näin ei aina ole, vaan jokaisesta palvelusta on vaihteleva määrä kopioita, joihin palveluun tuleva verkkoliikenne jaetaan. Luvussa 2.6.2 mainittiin, että mikropalveluita ajetaan yleensä konteissa. Palvelukopioita kannattaa ajaa omissa konteissa, jolloin ne ovat erillään toisistaan. Tämän avulla muut kopiot voivat jatkaa toimintaa ja pyyntöjen vastaanottamista, jos yhteen konteista tulee vika.

2.7.2 Kuormantasaus

Kun jonkin mikropalvelun tarvitsee tehdä pyyntö toiseen mikropalveluun, sillä ei yleensä tarkoiteta sitä, että pyyntö menisi aina vain yhteen palvelukopioon (Babal 2023, luku 5.1). Mikropalveluiden ajoympäristöissä on yleensä useampia identtisiä kopioita palvelusta, kuten edellisessä alaluvussa mainittiin. Pyynnöt näihin palvelukopioihin jaetaan käyttäen palvelinta, jota kutsutaan kuormantasaajaksi (Babal 2023, luku 5.1). Kuormantasaus on siis kuormantasaajien käyttöä pyyntöjen jakamiseksi useammalle eri palvelimelle. Kuormantasaajan ei aina tarvitse olla erillinen välityspalvelin, vaan pyynnön tekevä palvelu voi toimia myös kuormantasaajana. Kuormantasauksesta on olemassa kaksi pääasiallista variaatiota, joita mikropalveluympäristöissä käytetään.

Ensimmäinen näistä variaatioista on palvelinpuolen kuormantasaus. Siinä asiakkaana toimiva palvelu tekee pyynnön kuormantasaajalle, joka jakaa pyynnön jollekin saatavilla olevalle palvelukopiolle kutsuttavasta palvelusta. Pyynnön lähettäjän ei tarvitse olla tietoinen kutsuttavan palvelun kopioista. (Babal 2023, luku 5.1.1.) Sen täytyy ainoastaan tietää palvelun kuormantasaajan osoite. Tässä kuormantasauksessa kuormantasaaja sijaitsee kutsuvan palvelun ja kutsuttavien palvelukopioiden välissä. Kuormantasaaja toimii siis välityspalvelimena.

Toinen kuormantasauksen variaatio on asiakaspuolen kuormantasaus. Siinä asiakkaana toimiva palvelu on tietoinen kutsuttavan palvelun kopioista. Se valitsee jonkin algoritmin perusteella yhden saatavilla olevista palvelukopioista, johon se lähettää pyynnön. Joissain toteutuksissa kutsuttava palvelukopio voi palauttaa asiakkaalle raportin, jonka perusteella asiakas valitsee saatavilla olevista palvelukopioista yhden, johon seuraava pyyntö lähetetään. (Babal 2023, luku 5.1.2.) Tässä ei siis ole palveluiden välissä erillistä kuormantasaajapalvelinta. Kutsuva palvelu voi toimia itse kuormantasaajana.

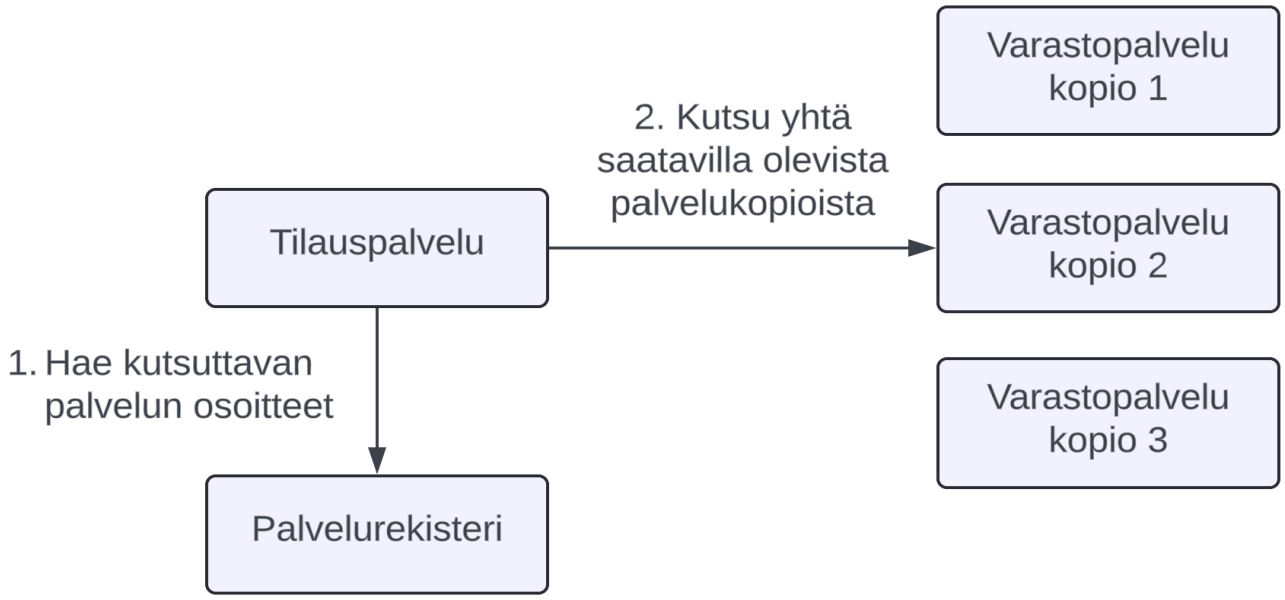
2.7.3 Palvelun löytäminen

Palvelun löytämisellä (englanniksi service discovery) tarkoitetaan mikropalveluissa menetelmää, jonka avulla voidaan automaattisesti havaita, rekisteröidä ja jakaa mikropalveluiden sijainteja tietoverkossa. Se on olennainen osa mikropalvelusovellusta, mikä mahdollistaa dynaamisen kommunikoinnin mikropalveluiden välillä. Tässä menetelmässä käytetään palvelurekisteriksi kutsuttua palvelinta (englanniksi service registry), joka rekisteröi kaikki mikropalvelusovelluksen palveluiden IP-osoitteet ja portit. Palvelut voivat käyttää tätä rekisteriä löytämään niiden palveluiden osoitteet, joiden kanssa niiden täytyy kommunikoida. Palvelurekisteri voi automaattisesti vastata koodimuutoksiin, jolloin palveluiden osoitteita voidaan joustavasti lisätä ja poistaa. (Broshar 23.3.2021.) Osoitteet ovat IP-osoitteiden ja porttien yhdistelmiä, eli pistokkeita, joista kerrottiin luvun 2.3 lopussa.

Tuotantoympäristössä palvelukopioiden IP-osoitteet ja portit voivat vaihtua tuntien tai jopa minuuttien välein. Moderneissa dynaamisissa mikropalveluympäristöissä tarvitaan automaattinen tapa ylläpitää eri palvelukopioiden osoitteita. Palvelurekisterin avulla tämä on mahdollista, jolloin rekisteri päivitetään joka kerta uuden palvelukopion tullessa tai poistuessa. (Kong 10.3.2022.)

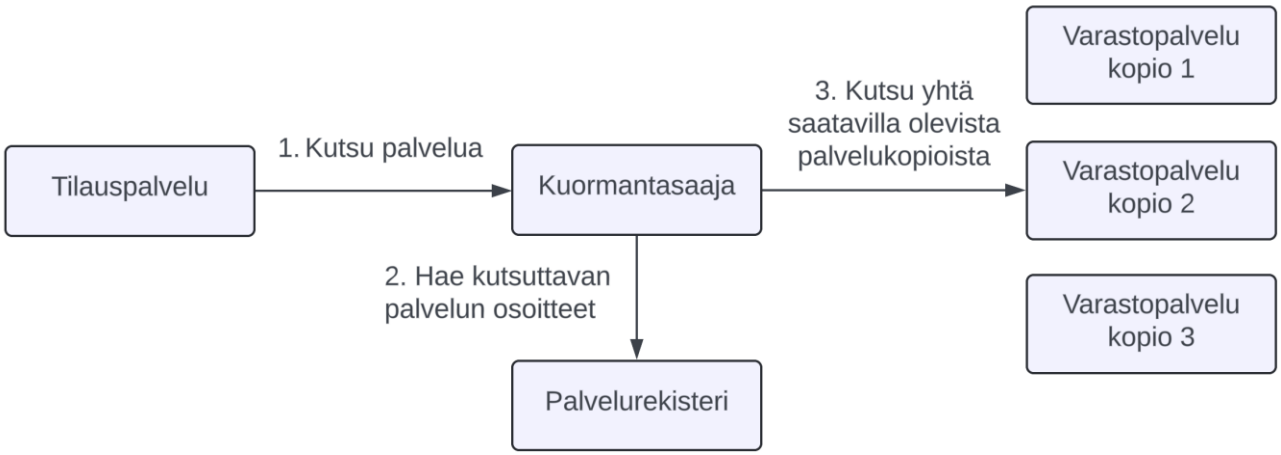
Kun mikropalvelut kommunikoivat keskenään synkronisesti esimerkiksi gRPC:n avulla, niiden täytyy tietää mihin osoitteeseen API-pyynnöt lähetetään. Yksi tapa on koodata tarvittavien mikropalveluiden osoitteet suoraan mikropalvelun lähdekoodiin, mutta tämä ei ole kannattava tapa useammastakin syystä. Jos mikropalveluiden osoitteet muuttuvat, ne täytyy päivittää lähdekoodiin. Tämä voi olla haastavaa ja aikaa vievää. Lisäksi jos mikropalvelut otetaan käyttöön pilvipalveluissa, pilvipalvelun tarjoaja tuottaa mikropalveluille uniikit osoitteet, jotka eroavat paikallisesta kehittäjän tietokoneella pyörivästä ympäristöstä. Tämä tuottaa ongelmia eri ympäristöjen välillä, minkä takia keskitetty palvelurekisteri on kannattavampi ratkaisu mikropalveluiden kommunikointiin. (Broshar 23.3.2021.)

Palvelun löytäminen voidaan toteuttaa kahdella eri tavalla. Ensimmäinen on asiakaspuoleinen ja toinen palvelinpuoleinen. Suoraviivaisemmassa asiakaspuolen palvelun löytämisessä mikropalvelut rekisteröivät itsensä käynnistyksen yhteydessä palvelurekisteriin, mihin niillä on tämän jälkeen suora yhteys. Palvelut, joiden tarvitsee yhdistää toiseen palveluun, voivat hakea palvelurekisteristä tarvittavan palvelun osoitteen jollakin kriteerillä, kuten palvelun nimellä. (Babal 2023, luku 2.4.) Tässä palvelu on kuitenkin vastuussa kuormantasauksesta, jolloin sen täytyy jakaa pyynnöt eri palvelukopioihin, jos niitä on enemmän kuin yksi. Jos pyyntöjä lähetettäisiin vain yhteen palvelukopioista, kyseinen palvelukopio ylikuormittuisi ja muita kopioita ei hyödynnettäisi ollenkaan. (Shuiskov 2022, luku 2 alaluku Client-side service discovery.) Kuva 6 havainnollistaa asiakaspuolen palvelun löytämistä.



Kuva 6. Palvelu kutsuu toista palvelua asiakaspuolen palvelun löytämisellä (mukaillen Shuiskov 2022, luku 2 alaluku Client-side service discovery)

Palvelinpuolen palvelun löytäminen on toinen tapa löytää palveluiden osoitteet, missä käytetään kuormantasaajaa. Kuormantasaaja on yhteydessä palvelurekisteriin ja hakee sieltä rekisteröityjen palvelujen osoitteet. Kun palvelun tarvitsee yhdistää toiseen palveluun, se hakee palvelun osoitteen kuormantasaajan kautta, eikä suoraan palvelurekisteristä. (Babal 2023, luku 2.4.) Kuormantasaaja jakaa pyynnöt haetun palvelun palvelukopioihin. Tämä on hyödyllistä pyyntöä tekevälle palvelulle, koska sen ei tarvitse olla tietoinen palvelurekisteristä, jolloin ohjelmalogiikasta tulee yksinkertaisempi. (Shuiskov 2022, luku 2 alaluku Server-side service discovery.) Kuvasta 7 nähdään, miten palvelinpuolen palvelun löytäminen toimii.



Kuva 7. Palvelu kutsuu toista palvelua palvelinpuolen palvelun löytämisellä (mukaillen Shuiskov 2022, luku 2 alaluku Server-side service discovery)

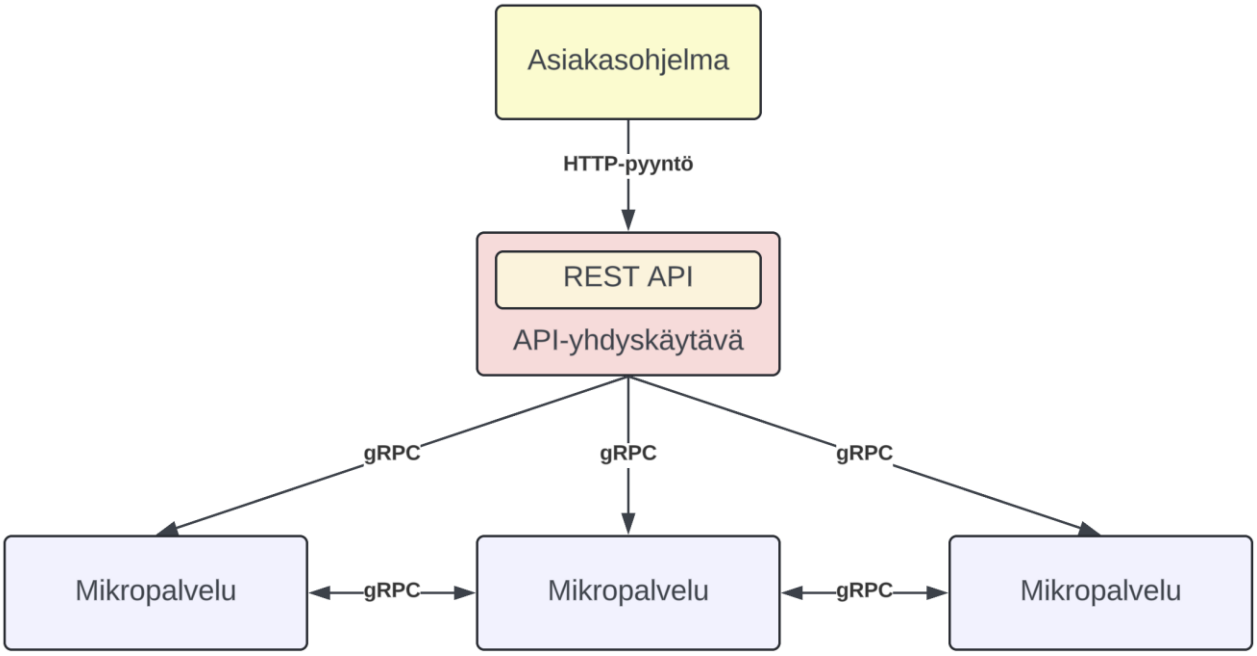
Palvelun löytämisellä voidaan siis mahdollistaa kommunikointi mikropalveluiden välillä. Palvelun löytäminen voidaan toteuttaa itse, mutta käyttöönottoympäristöissä, kuten mikropalveluihin soveltuvalla Kubernetes-alustalla, on sisäänrakennetut ominaisuudet palveluiden löytämiseksi (Babal 2023, luku 2.4). Käyttöönottoympäristöjen sisäänrakennetut ominaisuudet säästävät vaivannäöltä, jolloin palvelun löytämistä ei tarvitse itse toteuttaa ja ylläpitää (Kong 10.3.2022). Mikropalveluiden ajoympäristöön tarkoitetut alustat tarjoavat myös paljon muita komponentteja haasteiden helpottamiseksi. Esimerkiksi Kubernetes tarjoaa sisäänrakennettuna myös kuormantasauksen.

2.7.4 API-yhdyskäytävä

API-yhdyskäytävä on palvelu, joka toimii mikropalvelusovelluksen sisääntulopisteenä sovelluksen ulkopuolisille asiakasohjelmille. Se on vastuussa toiminnoista, kuten autentikoinnista, monitoroinnista, käytönrajoituksesta, protokollien kääntämisestä ja pyyntöjen ohjaamisesta oikeisiin mikropalveluihin. Kaikki mikropalvelusovellukseen tulevat pyynnöt menevät ensin API-yhdyskäytävään, joka ohjaa pyynnöt oikeisiin palveluihin. API-yhdyskäytävä kokoaa palveluilta saadut vastaukset yhdeksi vastaukseksi, jonka se palauttaa API-yhdyskäytävään tulleelle pyynnön lähettäjälle. API-yhdyskäytävä voi myös kääntää pyyntöjen käyttämiä protokollia sovelluksen ulkopuolisen asiakasohjelman ja mikropalveluiden välillä. (Richardson 2018, luku 8.2.1.) API-yhdyskäytävä on siis olennainen osa mikropalvelusovellusta. Tietoliikenne kulkee sen kautta sovelluksesta sisään ja sieltä ulos.

API-yhdyskäytävä voidaan toteuttaa itse, mutta on olemassa myös valmiita API-tuotteita, joiden avulla voidaan pystyttää API-yhdyskäytävä helpommin ja tietoturvallisemmin. Monia niistä voidaan käyttää pilvipalveluiden kautta, jolloin ei tarvitse hallita omia palvelimia ja infrastruktuuria niiden ajamiseksi. Suosittuja API-yhdyskäytäviä ovat esimerkiksi Kong Gateway, Amazon API Gateway ja Azure API Management.

Protokollien kääntäminen on yksi API-yhdyskäytävän hyödyistä mikropalveluiden kommunikoinnissa. Sen avulla voidaan esimerkiksi kääntää HTTP-protokollat gRPC:hen. Asiakasohjelma voi lähettää pyynnön HTTP-protokollalla API-yhdyskäytävään, joka ohjaa pyynnön mikropalveluille gRPC-muodossa. Tämä parantaa mikropalvelusovelluksen sisäistä kommunikointia ja antaa asiakasohjelmille, kuten selainohjelmille, helppokäyttöisemmän API:n ja paremman tuen. Kuva 8 havainnollistaa tätä API-yhdyskäytävän kommunikointiprotokollien käännösprosessia.



Kuva 8. Kommunikointiprotokollien kääntäminen API-yhdyskäytävän avulla

2.8 Mikropalveluiden integroiminen API:lla

Tähän asti on käyty läpi mikropalveluiden toimintaa ja miten niitä ajetaan. Ei kuitenkaan riitä, että niitä vain ajetaan palvelimilla. Ne täytyy myös integroida yhteen, jotta niihin ohjatut toiminnot saadaan suoritettua, ja jotta ne toimivat yhtenä eheänä järjestelmänä. Mikropalveluista ei ole kauheasti hyötyä, ellei niiden kanssa pysty kommunikoimaan. API on mekanismi, jolla tämä ongelma saadaan ratkaistua.

2.8.1 API:n toiminta

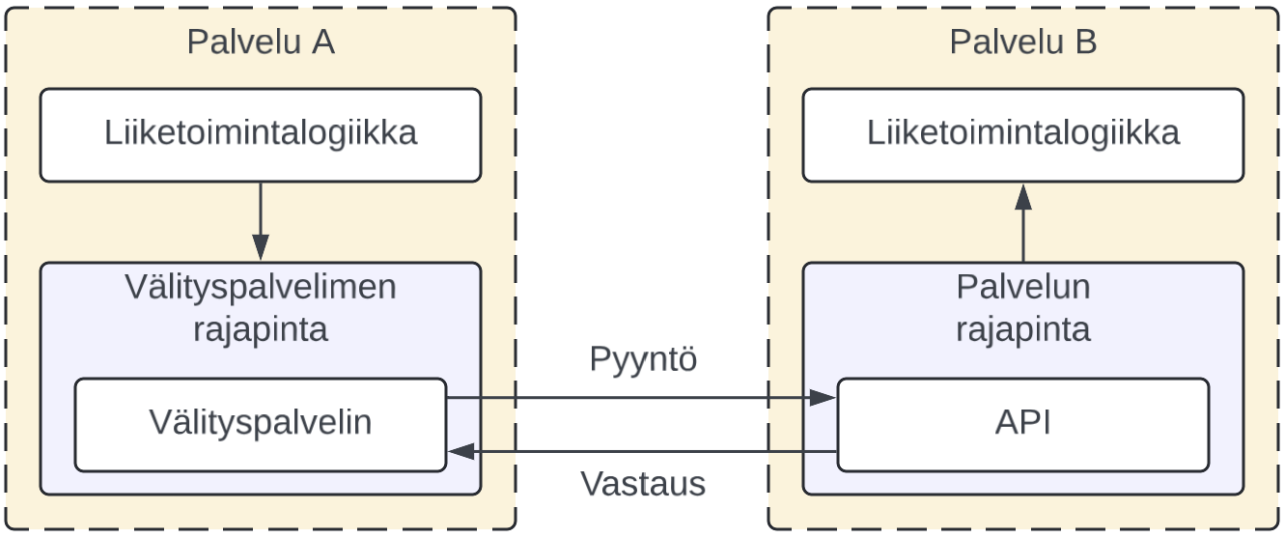
Mikropalvelut voivat kommunikoida keskenään API:en kautta verkon yli. API toimii välittäjänä kahden palvelun välillä mahdollistaen pyyntöjen ja vastausten lähettämisen palvelusta toiseen. Se yhdistää kaksi tietokonetta tai tietokoneohjelmaa rajapinnan kautta. API:a ei tule sekoittaa käyttöliittymän kanssa. API yhdistää ohjelmistoja ja tietokoneita toisiinsa, eikä loppukäyttäjän ole tarkoitus käyttää sitä. (Acharya 22.6.2022.) Ohjelmistokehittäjät ohjelmoivat API:n ja integroivat sen kehitettävään ohjelmistoon. Tämän jälkeen ohjelmistot tai tietokoneet voivat käyttää tätä API:a kommunikoidakseen keskenään API:n määräämien sääntöjen mukaan.

Kun mikropalveluita yhdistetään API:lla, se määrittelee säännöt, joiden mukaan palvelut kommunikoivat keskenään (Hillpot 20.5.2024). Säännöt pitävät sisällään käytettävän protokollan ja tiedonsiirtomuodon. Lisäksi API seuraa jotakin tiettyä API-spesifikaatiota, jonka mukaan API:a tulisi käyttää tai rakentaa. (Acharya 22.6.2022.) API pitää sisällään toimintoja, joita toinen palvelu voi kutsua

toimintojen suorittamiseksi. Toimintojen nimet vaihtelevat API:n tyypin mukaan. Toimintoja voidaan kutsua esimerkiksi pyynnöiksi, metodeiksi, päätepisteiksi tai aliohjelmiksi (Acharya 22.6.2022). API on keskeinen asia ohjelmistokehittäjille, koska sen kautta ollaan vuorovaikutuksessa datan kanssa. Hyvin suunniteltu API on helppo käyttää eikä se ole hämmentävä. (Hillpot 20.5.2024.)

Jokaisella mikropalvelulla on tyypillisesti oma API, jonka kautta toinen palvelu voi esimerkiksi hakea dataa siltä. API:ssa on päätepisteitä, joihin toinen palvelu tekee verkkokutsuja. Verkkokutsun mukana voidaan lähettää dataa ja kutsun lähetettyä siihen voidaan saada vastaus. Vastauksen mukana voidaan myös lähettää dataa. Monesti API-kutsujen vastauksien yhteydessä halutaan myös palauttaa tilakoodi, josta selvenee, onko API-kutsu onnistunut vai epäonnistunut. Kutsu voi epäonnistua esimerkiksi verkkovian takia, joten halutaan jokin tapa, jolla voidaan tarkistaa se API-kutsun lähettäneen ohjelmiston ohjelmakoodin logiikassa.

Kuva 9 havainnollistaa palveluiden kommunikointia API:n kautta. Palvelun A liiketoimintalogiikka kutsuu välityspalvelimen rajapintaa, joka lähettää pyynnön palvelun B API:iin. API kutsuu liiketoimintalogiikkaa, jonka jälkeen se palauttaa vastauksen palvelulle A. Palvelu A käsittelee vastauksen omassa liiketoimintalogiikassa. Välityspalvelimen rajapinta sisältää tiedon API:n päätepisteistä, joita voidaan kutsua. Sen avulla API-kutsut voidaan siis laukaista. (Kuva 9.) Välityspalvelimen rajapinta yleensä sisältää myös käytettävän kommunikointiprotokollan (Richardson 2018, luku 3.2).



Kuva 9. Palvelu kutsuu toista palvelua API:n kautta (mukaillen Richardson 2018, luku 3.2)

2.8.2 Suosituimpia API-tyyppejä

Suosituimpia API-tyyppejä ovat olleet RPC API, SOAP API ja REST API. RPC API:t perustuvat etäproseduurikutsuihin ja ovat aikaisimpia API-tyyppejä. Ne ovat suunniteltu suorittamaan ohjelmakoodia eri palvelimilla. (Acharya 22.6.2022.) RPC kehitettiin 1970-luvun lopussa ja 1980-luvun alussa. RPC API:sta on monia eri toteutuksia, joista nykyään yksi suosituimmista on gRPC. (Amazon Web Services 2024b.) gRPC on viitekehys, joten se tarjoaa pelkän RPC API:n lisäksi myös lisäominaisuuksia, joita voidaan hyödyntää kehityksen aikana. gRPC:tä käsitellään tarkemmin tämän opinnäytetyön luvussa 4. Yksinkertaistettuna RPC API tarjoaa funktioita, joita erillinen prosessi voi kutsua esimerkiksi toisella palvelimella.

SOAP API:t perustuvat XML-pohjaiseen ohjelmointiin. Ne tarjoavat korkean tietoturvatason ja niitä käytetään usein rahoituspohjaisissa tietojärjestelmissä. (Acharya 22.6.2022.) SOAP API:a voidaan pitää hieman vanhentuneena, koska uutta API:a harvoin toteutetaan enää sillä. SOAP kehitettiin ennen REST:iä, ja REST nykyään ratkookin paljon SOAP:iin liittyviä ongelmia. SOAP API:a käytetään kuitenkin vielä paljon vanhemmissa tietojärjestelmissä. (Amazon Web Services 2024a.)

REST API on yksi suosituimmista API-tyypeistä API:en rakentamiseen. REST on suosittu, koska se käyttää HTTP-metodeja, jotka ovat tuttuja valtaosalle ohjelmistokehittäjistä, ja jotka ovat helppoja käyttää. (Hillpot 20.5.2024.) REST on arkkitehtuurityyli API:n rakentamiseen, jossa toiminnot perustuvat resursseihin, joita käytetään URI:en avulla. Resursseja esitetään yleensä JSON- tai XML-datamuodossa, jotka siirretään asiakkaan ja palvelimen välillä HTTP-pyynnöissä ja -vastauksissa. (The Postman Team 20.11.2023.) REST:illä on erinomainen tuki eri asiakasohjelmille, kuten mobiilisovelluksille ja selainohjelmille. Tämän takia se sopii etenkin moniin API-yhdyskäytäviin. REST on yksinkertainen ja saattaa olla rajoittava monimutkaisempiin tai edistyneempiin kommunikointitarkoituksiin.

API:t voidaan jakaa julkisiin ja yksityisiin. Julkista API:a käytetään, kun mikropalvelua käyttävän ulkoisen asiakasohjelman tarvitsee tehdä pyyntö palveluun. Ulkoisena asiakasohjelmana voi toimia esimerkiksi selainohjelma tai mobiilisovellus. Yksityistä API:a taas käytetään palveluiden väliseen kommunikointiin. Palveluiden välisessä kommunikoinnissa pitää huomioida paremmin suorituskyky. (Ozkaya 7.9.2021.) Yksityinen API on tarkoitus olla saatavilla mikropalveluiden kehittäjille ja mikropalvelusovelluksen sisäisille palveluille, eikä sovelluksen ulkopuolisille asiakasohjelmille.

2.8.3 Datan sarjoittaminen osana tiedonsiirtoa

Datan sarjoittaminen on olennainen osa mikropalveluiden kommunikointia. Se on menetelmä, jossa datan muoto muunnetaan toiseen muotoon, jossa se voidaan siirtää tai varastoida. Myöhemmin se voidaan muuntaa takaisin alkuperäiseen muotoon käsittelyä varten. Datan sarjoittamisen pääasiallinen käyttötapaus mikropalveluissa on datan siirto niiden välillä, jolloin sarjoittamismuoto toimii niiden yhteisenä kielenä. (Shuiskov 2022, luku 4 alaluku The basics of serialization.)

Mikropalveluiden välinen kommunikointi aiheuttaa paljon verkkoliikennettä. Tästä syystä tarvitaan mahdollisimman hyvä suorituskyky, jotta sovellus toimisi mahdollisimman nopeasti ja viiveettömästi. Datan sarjoittamisnopeus ja siirrettävän datan koko on tärkeä osa tätä. (Ozkaya 7.9.2021.)

Suosittu REST-arkkitehtuurityyli käyttää yleensä JSON:ia tai XML:ää sen sarjoittamismuotona. Nämä ovat tekstipohjaisia, mikä tekee niiden käyttämisestä helpompaa. (The Postman Team 20.11.2023.) Datan sarjoittamisessa mikropalvelu muuntaa lähetettävän viestin muodon ohjelmointikielen tietorakenteesta esimerkiksi JSON-muotoon. Viesti lähetetään tekstimuotoisena JSON:ina toiseen palveluun, jossa se puretaan takaisin ohjelmointikielen tietorakenteeseen. Tämä vaikuttaa melko yksinkertaiselta ja helpolta tavalta, mutta se ei ole paras vaihtoehto palveluiden väliseen kommunikointiin. Tekstipohjaisen muotonsa takia JSON-datan koko on melko iso tavuissa, jolloin sen muuntamisnopeus ei ole paras mahdollinen verrattuna tiettyihin sarjoittamismuotoihin. (Shuiskov 2022, luku 4 alaluku The basics of serialization.)

Muita suosittuja sarjoittamismuotoja JSON:in ja XML:än lisäksi ovat esimerkiksi YAML, Apache Thrift, Apache Avro ja Protocol Buffers. Kaikki sarjoittamismuodot eivät sovellu mikropalveluiden kommunikointiin, kuten YAML. YAML:ia käytetään usein konfiguraatiotiedostojen sarjoittamismuotona ja sen käyttö palveluiden kommunikoinnissa on harvinaista. (Shuiskov 2022, luku 4 alaluku Popular serialization formats.) Protocol Buffers taas on tehokas ja erittäin soveltuva palveluiden väliseen kommunikointiin. gRPC käyttää sitä pääasiallisesti. Protocol Buffersista kerrotaan tarkemmin luvuissa 4.2.1 ja 4.6.1.

3 Mikropalveluiden kommunikointitavat

Mikropalveluiden välinen kommunikointi voidaan toteuttaa monilla eri tavoilla ja teknologioilla, eikä siihen ole aina yhtä oikeaa ratkaisua. Laajan valikoiman takia kehittäjät voivat monesti ylikuormittua valinnoista. (Newman 2021, luku 4 alaluku Technology for Inter-Process Communications: So many Choices.) On kuitenkin hyvä olla tietoinen tarjolla olevista kommunikointitavoista ennen kuin mikropalveluiden kommunikointia aletaan toteuttamaan, koska kommunikointitapa on hyvin keskeinen osa sitä.

3.1 Synkroninen ja asynkroninen kommunikointitapa mikropalveluissa

Synkroninen ja asynkroninen kommunikointi ovat kaksi pääasiallisinta kommunikointitapaa mikropalveluissa (Ozkaya 7.9.2021). Synkronista kommunikointia voidaan pitää perinteisenä asiakas-palvelinmallina, jossa asiakasohjelma lähettää pyynnön palvelimelle ja odottaa siltä vastausta. Vastauksen saatua ohjelman suorittaminen voi jatkua. Tämän tyyppinen kommunikointi voidaan toteuttaa API:en avulla, kuten esimerkiksi REST:illä, missä mikropalvelu lähettää verkkopyynnön toiselle mikropalvelulle. Palvelu käsittelee pyynnön ja lähettää sen jälkeen vastauksen takaisin pyynnön lähettäjälle. (Sundar 21.7.2023.) Vastausta tarvitaan tilanteissa, joissa tarvitaan tehdä jokin toiminto vastauksen perusteella tai muuten vain halutaan varmistaa pyynnön onnistuminen (Newman 2021, luku 4 alaluku Pattern; Synchronous Blocking). Synkroninen kommunikointi ei kuitenkaan tuki koko tietokoneprosessia, jos siinä käytetään useampia eri säikeitä. Vastausta odotetaan ainoastaan säikeessä, jossa pyynnön lähettävä ohjelmakoodi suoritettiin. Monisäikeiset prosessit ovat monimutkaisia, eikä niitä ole tarkoitus tässä työssä käsitellä.

Asynkroninen kommunikointi toimii eri tavalla. Siinä asiakasohjelma lähettää pyynnön palvelimelle, mutta se ei odota välitöntä vastausta. Palvelin käsittelee pyynnön, mutta se lähettää vastauksen takaisin vasta myöhemmin tai ei ollenkaan. Tämän tyyppisessä kommunikoinnissa käytetään usein viestinvälittäjiä, kuten Apache Kafkaa tai RabbitMQ:ta. (Sundar 21.7.2023.) Viestinvälittäjät toimivat julkaisu-tilausmekanismilla, jossa palvelut voivat julkaista viestejä viestijonoon ja toiset palvelut käyttävät tähän viestijonoon tulleita viestejä joko heti tai myöhemmin (Ozkaya 7.9.2021). Tätä mekanismia pidetään tapahtumiin pohjautuvana kommunikointina, jossa mikropalvelu voi lähettää viestejä viestijonoon muiden palveluiden käsiteltäväksi eri tapahtumien yhteydessä. Tapahtuma on jokin asia, joka on yleensä tapahtunut viestiä lähettävän mikropalvelun sisällä. Tässä kommunikointitavassa palvelun ei tarvitse olla tietoinen muista palveluista. (Newman 2021, luku 4 alaluku Pattern: Event-Driven Communication.) Asynkronista kommunikointia käytetään yleensä tapahtumiin pohjautuvassa kommunikoinnissa, mutta sitä voidaan käyttää myös pyyntöihin ja vastauksiin pohjautuvassa kommunikoinnissa (Newman 2021, luku 4 alaluku Styles of Microservice Communication).

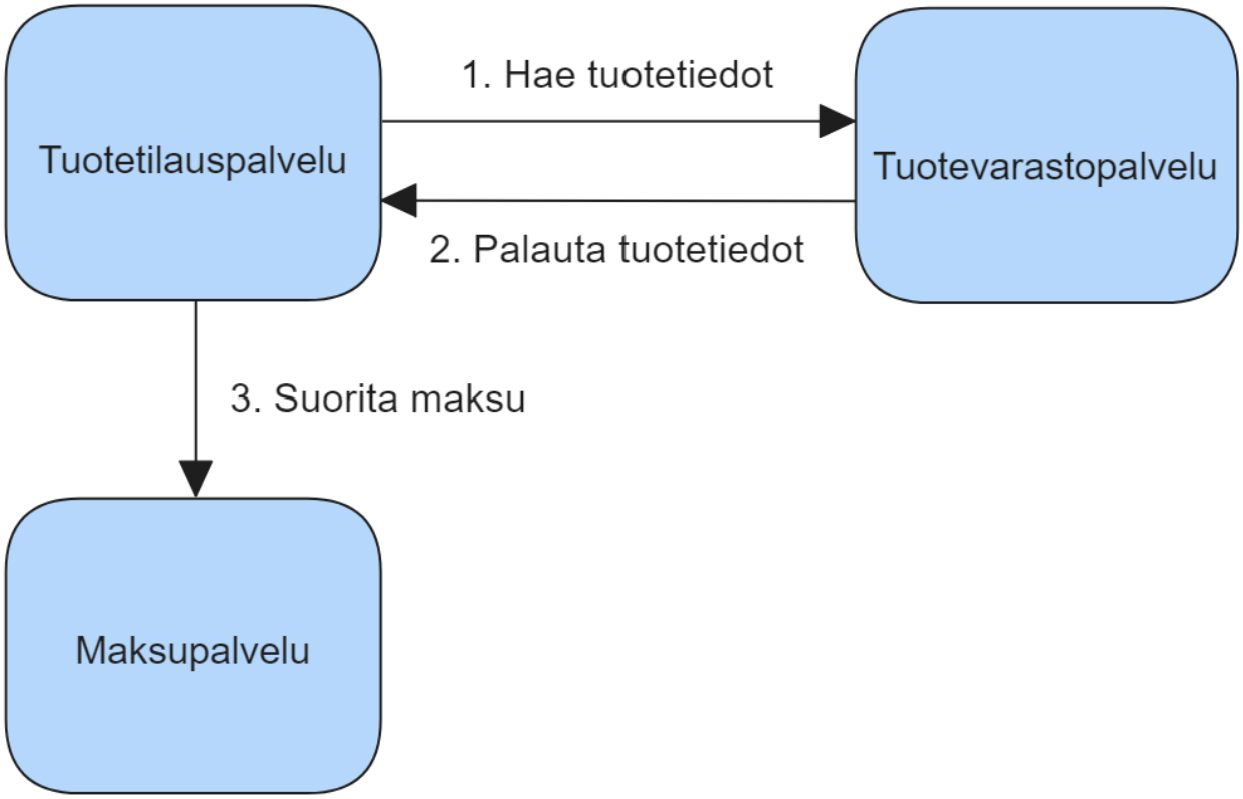
Mikropalvelusovelluksissa harvoin yhdistetään kaikkia mikropalveluita yhdellä kommunikointitavalla. Yleensä käytetään synkronisen ja asynkronisen kommunikoinnin yhdistelmiä. (Ozkaya 7.9.2021.) On myös yleistä, että yksittäinen mikropalvelu käyttää useita eri kommunikointitapoja. Palvelulla voi olla esimerkiksi API synkronista kommunikointia varten, jolloin se voi käsitellä muilta palveluilta tulevia pyyntöjä. Sen lisäksi se voi välittää viestejä viestijonoon muiden palveluiden käsiteltäväksi jonkin tapahtuman yhteydessä. (Newman 2021, luku 4 alaluku Styles of Microservice Communication.)

Asynkroninen kommunikointi on laaja aihealue ja monimutkaisempi kommunikointitapa kuin perinteisempi synkroninen kommunikointi. Siitä on olemassa monia eri toteutuksia, joissa täytyy yleensä huomioida enemmän asioita kuin synkronisessa kommunikoinnissa. Synkroninen kommunikointi on yksinkertaisempi ja monesti suoraviivaisempi toteuttaa. Synkronista kommunikointia näkee paljon muissakin ohjelmistoissa kuin mikropalveluissa ja se on varmasti monelle ohjelmistokehittäjälle tuttu, jos on kehittänyt ohjelmistoja jo jonkin aikaa.

3.2 Synkronisen pyyntöihin ja vastauksiin pohjautuvan kommunikoinnin toiminta

Pyyntöihin ja vastauksiin pohjautuvassa kommunikoinnissa mikropalvelu lähettää pyynnön toiselle palvelulle pyytäen sitä tekemään jonkin tietyn toiminnon. Toiminnon suorittamisen jälkeen palautetaan vastaus takaisin pyynnön lähettäjälle. Pyyntöjen ja vastauksien mukana voidaan lähettää myös dataa. Esimerkkinä voidaan pitää tilannetta, jossa mikropalvelun tarvitsee saada dataa toiselta palveluta. Data palautetaan pyynnön vastauksen mukana. Tämän kaltainen vuorovaikutus voidaan toteuttaa joko synkronisesti tai asynkronisesti. (Newman 2021, luku 4 alaluku Pattern: Request-Response Communication.)

Kuvassa 10 on havainnollistettuna kolmen mikropalvelun välinen kommunikointi käyttäen synkronista pyyntöihin ja vastauksiin pohjautuvaa kommunikointitapaa. Siinä tuotetilauspalvelu lähettää pyynnön tuotevarastopalvelulle pyytäen sitä palauttamaan tilauksen tuotteiden tiedot. Vastauksena pyyntöön tuotetilauspalvelu saa tuotevarastopalvelulta tilaukseen liittyvät tuotetiedot. Vastauksen jälkeen tuotetilauspalvelu lähettää pyynnön maksupalvelulle pyytäen sitä käsittelemään maksun käyttäen haettuja tuotetietoja. (Kuva 10.) Tämä tapahtuma on yksinkertainen esimerkki mikropalveluiden synkronisesta kommunikoinnista, missä asiakasohjelmana toimiva mikropalvelu odottaa vastausta palvelimena toimivalta mikropalvelulta. Oikea järjestelmä on usein paljon monimutkaisempi.



Kuva 10. Pyyntöihin ja vastauksiin pohjautuva synkroninen kommunikointi kolmen mikropalvelun välillä (mukaillen Newman 2021, luku 4 alaluku Pattern: Request-Response Communication)

Synkronisiin pyyntöihin ja vastauksiin pohjautuvassa kommunikoinnissa on yleistä käyttää HTTP-protokolliin ja REST-arkkitehtuuriin pohjautuvia lähestymistapoja. Ne ovat yleisimpiä tapoja suunnitella API mikropalvelulle, etenkin jos mikropalvelu halutaan tehdä saatavaksi mikropalvelusovelluksen ulkopuolelle. Mikropalvelusovelluksen sisäisessä kommunikoinnissa voidaan kuitenkin hyötyä binäärimuodossa tapahtuvasta kommunikoinnista. Esimerkiksi gRPC:n avulla tämä on mahdollista. (Ozkaya 7.9.2021.) Binäärimuodon hyödyistä kerrotaan myöhemmin luvussa 4.2.1.

Synkronisessa kommunikoinnissa palvelun täytyy olla tietoinen toisesta palvelusta pyyntöjen lähettämisen yhteydessä. Tämä on yksi synkronisen kommunikoinnin haasteista asynkroniseen kommunikointiin verrattuna. Joka kerta, kun palvelun tarvitsee tehdä synkroninen pyyntö toiseen palveluun, on olemassa riski, että pyyntö epäonnistuu tai toinen palvelu palauttaa vastauksen pitkän viiveen jälkeen (Richardson 2018, luku 3.2.3). Virhetilanteiden sattuessa, kuten toisen palvelun ollessa pois käytöstä, täytyy olla valmiina tarkka suunnitelma ja ohjelmalogiikka virheiden käsittelyä varten (Newman 2021, luku 4 alaluku Synchronous Blocking). Virheiden sattuessa voidaan siis esimerkiksi yrittää pyyntöä uudestaan. Tähän on olemassa useita eri tapoja ja tekniikoita, joista osa on hyvin edistyneitä. Näihin ei ole tarkoitus keskittyä tässä opinnäytetyössä.

3.3 Synkronisen kommunikointitavan valinta

Newman painostaa oikean teknologian valitsemisen tärkeyttä prosessien välisessä kommunikoinnissa. Hänen mukaansa on tärkeää ensin pohtia, millaista kommunikointitapaa tarvitsee ja sen jälkeen vasta valita oikea teknologia valittuun kommunikointitapaan. (Newman 2021, luku 4 alaluku Technology for Inter-Process Communication: So Many Choices.) Valittu kommunikointitapa riippuu käyttötarkoituksesta (Indrasiri & Kuruppu 2020, luku 1).

Yksinkertaisimmissa tapauksissa valitaan monesti synkroninen kommunikointi, koska se on helpoin toteuttaa. Sillä voi olla myös hyvä lähteä liikkeelle, jos muut haastavammat kommunikointitavat tuottavat kehitystiimeille ongelmia. Synkroninen kommunikointi sopii tilanteisiin, joissa palveluiden täytyy suoraan lähettää pyyntöjä toisiin palveluihin, ja joissa palvelut tarvitsevat pyyntöihin suoran vastauksen. Tässä ei siis tarvita erillisiä viestinvälittäjiä tai viestijonoja palveluiden välille, mikä vähentää toteutuksen monimutkaisuutta.

Joissain tilanteissa synkroninen kommunikointi saattaa myös olla ainoa vaihtoehto, jos asynkroninen kommunikointi ei auta ratkottavassa ongelmassa. Esimerkiksi operaatio, joka jakautuu useaan eri palveluun, saattaa tarvita synkronista kommunikointia toimiakseen. Esimerkkitilanteessa yhden palvelun täytyy kutsua toista palvelua ja saada siltä vastaus ennen kuin operaatio voi edetä seuraaviin palveluihin. Tämän avulla saavutetaan johdonmukainen kommunikointiketju. Siinä voidaan myös esimerkiksi kumota edellisten palveluiden tietokantoihin tehtyjä muutoksia, jos operaatio epäonnistuu kommunikointiketjun seuraavassa kutsuttavassa palvelussa odottamattoman vian takia.

Kommunikointitavan valinnalla on usein seurauksia valmiissa järjestelmässä, joten se tulee valita harkiten. Kommunikointitapa ei ole sidottu mihinkään teknologiaan, joten sen miettiminen ennen teknologian valintaa on hyödyllistä vaatimusten määrittelyssä. (Richardson 2018, luku 3.1.1.) Kun on valittu sopiva kommunikointitapa mikropalveluihin, voidaan valita jokin teknologia valittuun kommunikointitapaan. Tässä on tärkeää olla tietoinen tarjolla olevista vaihtoehdoista. Synkroniseen mikropalveluiden kommunikointiin voidaan valita gRPC, koska se on suunniteltu tähän käyttötarkoitukseen ja se antaa monia hyötyjä siihen. Lisäksi se on suosittu ja käytetty teknologia mikropalveluissa, johon monet ohjelmistokehittäjät luottavat.

4 Mikropalveluiden synkroninen kommunikointi gRPC:llä

Luvussa 2.8 käytiin läpi API:n tärkeyttä mikropalveluiden kommunikoinnissa, sekä suosituimpia API-tyyppejä. API mahdollistaa palveluiden synkronisen kommunikoinnin. Palvelun API voidaan toteuttaa monilla eri tekniikoilla kelvolliseksi ja toimivaksi, jolloin mikropalvelut voivat kommunikoida toistensa kanssa. gRPC:llä tämä voidaan kuitenkin toteuttaa erinomaisesti. gRPC tekee mikropalveluiden välisestä kommunikoinnista nopeampaa ja tehokkaampaa (Kuruppu 31.8.2021).

4.1 Taustatietoa gRPC:stä

gRPC on prosessien väliseen kommunikointiin käytettävä teknologia, jonka avulla voidaan operoida hajautettuja järjestelmiä kuin tekisi paikallisia funktiokutsuja niiden prosesseihin (Indrasiri & Kuruppu 2020, luku 1 alaluku What is gRPC?). gRPC:n avulla voidaan siis suorittaa toimintoja mikropalveluiden välillä. Toiminnot suoritetaan verkkokutsuilla etäproseduurikutsuja käyttäen. Etäproseduurikutsussa asiakasohjelma kutsuu ohjelmakoodin funktiota palvelimella kuin funktio olisi paikallinen funktio asiakasohjelman puolella (Amazon Web Services 2024b).

Kommunikointiprotokollan lisäksi gRPC on viitekehys, jossa on sisäänrakennettuna paljon ominaisuuksia etenkin mikropalveluiden kehitykseen. Näitä ominaisuuksia voidaan hyödyntää, kun mikropalveluiden kommunikointia ohjelmoidaan. Siitä löytyy esimerkiksi kuormantasaus ja palveluiden terveydentilan tarkistukset. (Babal 2023, luku 2.5.) Lisäksi gRPC:llä on mahdollista salata palveluiden välinen kommunikointi käyttäen TLS-protokollaa. Tämä on sisäänrakennettuna monien ohjelmointikielien gRPC-kirjastoissa.

gRPC on alun perin Googlen luoma projekti, joka julkaistiin avoimena lähdekoodina vuonna 2015. Ennen tätä Google käytti omaa sisäistä etäproseduurikutsuihin pohjautuvaa teknologiaansa nimeltä Stubby, jolla yritys yhdisti lukuisat mikropalvelunsa yli kymmenen vuoden ajan. Google kuitenkin päätti tehdä seuraavan version Stubbysta ja tuloksena syntyi gRPC. (gRPC Authors 2024.) Nykyään gRPC:tä käyttävät tietojärjestelmissään useat maailman suurimmat yritykset, kuten Netflix, Cisco ja Uber (Kong 27.4.2022).

gRPC:stä on vuosien aikana muodostunut suosittu projekti ja sille on olemassa useita aliprojekteja, kuten gRPC Gateway ja gRPC-Web. Se on aktiivisessa kehityksessä ja sille voidaan odottaa tulevan vielä monia hyödyllisiä parannuksia ja työkaluja. Koska lähdekoodi on avointa, siihen voidaan luottaa paremmin kuin suljetun lähdekoodin projekteihin. Lisäksi vapaaehtoiset kehittäjät voivat osallistua sen kehitykseen, jolloin esimerkiksi vikojen ja haavoittuvuuksien löytäminen tehostuu.

4.2 gRPC:n hyödyt

gRPC protokollana ja viitekehyksenä antaa monia hyötyjä. Osa näistä hyödyistä on saatavilla oletuksena, jolloin lisätoimenpiteitä tarvitaan hyvin minimaalisesti. On hyvä kuitenkin huomata, että gRPC:tä voidaan käyttää monella eri tapaa. Sen tuomat hyödyt siis vaihtelevat sen käyttötavan perusteella. gRPC on laaja teknologia, joten sen tunteminen syvällisesti on tärkeää sen hyödyntämisessä parhaalla mahdollisella tavalla.

4.2.1 Suorituskyky

gRPC-protokollalla on erinomainen suorituskyky. Tähän on kaksi pääasiallista syytä. Ensimmäinen on gRPC:n käyttämä HTTP/2-protokolla ja toinen taas datan sarjoittamismuotona käytettävä Protocol Buffers. Nämä molemmat ovat käytössä gRPC-protokollassa oletuksena. (Babal 2023, luku 1.1.1.) HTTP/2 auttaa gRPC:tä vähentämään rajoitteita, joita esimerkiksi REST:in yleensä käyttämässä HTTP/1.1-protokollaversiossa esiintyy. HTTP/2 tukee useampien HTTP-pyyntöjen lähettämistä samalla yhteydellä. Lisäksi se lähettää pyynnöt binäärimuodossa ja tiivistää HTTP-otsikot pienempään kokoon. Nämä vähentävät lähetettävän datan kokoa ja kommunikoinnin viivettä, tehden palveluiden kommunikoinnista nopeampaa ja kannattavampaa. (Mohan 19.7.2021.)

Uutta yhteyttä ei tarvite gRPC:n avulla joka kerta avata pyyntöä lähetettäessä, mikä nopeuttaa kommunikointia. Sen avulla on mahdollista ylläpitää yhteyksiä, joita voidaan käyttää uudelleen eri pyyntöihin yhteyden perustamisen jälkeen. Tästä on paljon hyötyä järjestelmissä, joissa pyynnöt kulkevat koko ajan palvelusta toiseen.

Toinen syy gRPC:n hyvään suorituskykyyn on Protocol Buffers. Protocol Buffers käyttää binäärimuotoa datan siirrossa palveluiden välillä, mikä nopeuttaa tiedonsiirtoa. Binäärimuoto on tiiviimpi ja tietokoneelle helpompi jäsentää kuin tekstipohjaiset muodot. (Broshar 27.4.2024.) Binäärimuodossa oleva data on erittäin tehokasta etenkin suurten datamäärien lähettämisessä (Mohan 19.7.2021). Nämä hyödyt eivät pelkästään tee järjestelmästä nopeampaa, mutta myös odotettavasti vähentävät tietoliikenteestä aiheutuvia kuluja organisaatioille, koska lähetettävän datan koko on pienempi binäärimuodon ansiosta. Tästä on erittäin paljon hyötyä, jos dataa lähetetään paljon, ja jos tietoliikenteestä maksetaan käytön perusteella.

4.2.2 Ohjelmakoodin automaattinen generointi

Viitekehyksenä gRPC tarjoaa sisäänrakennettuna ohjelmakoodin generoinnin asiakaspuolelle ja palvelinpuolelle. Kun gRPC API on määritelty prototiedostoihin Protocol Buffersilla, Protocol Buffersin kääntäjällä voidaan automaattisesti generoida näistä tiedostoista ohjelmakoodit monille eri ohjelmointikielille, joiden avulla voidaan ohjelmoida gRPC API. Tämä ohjelmakoodin generointi tekee API-kehityksestä luotettavampaa ja sujuvampaa. Muissa API-arkkitehtuurityyleissä, kuten REST:issä, joutuu käyttämään ylimääräistä kolmannen osapuolen työkalua tähän. (Amazon Web Services 2024c.)

Koska Protocol Buffers käyttää vahvaa tyypitystä API:en määrittelemiseen, generoidut gRPC-ohjelmakoodit vähentävät kehityksen aikana tapahtuvia ohjelmavirheitä. Sen avulla voidaan esimerkiksi määritellä, onko pyyntöviestin sisältämä kenttä tietotyypiltään tekstiä vai kokonaisluku. Kokonaislukujen osalta voidaan määritellä, onko se 32- vai 64-bittinen. Lisäksi ohjelmakoodin generointi auttaa asiakasohjelman ja palvelinohjelman välistä vuorovaikutusta, koska molemmat perustuvat yhteen yhteiseen API-määritelmään, josta voidaan helposti tarkastaa kenttien käyttämät tietotyypit. Protocol Buffersin ansiosta tähän API-määritelmään voidaan myös suoraan kirjoittaa dokumentaatiokommentteja esimerkiksi viestien kentistä, jolloin erillistä dokumentaatiota ei välttämättä tarvita. Nämä dokumentaatiokommentit siirtyvät koodigeneroinnin yhteydessä myös generoituihin ohjelmakoodeihin. REST:in yleensä käyttämät JSON ja XML eivät anna samanlaista tyyppiturvallisuutta ja tehokasta API-määritelmää, mikä voi tehdä niiden käyttämisestä isoissa järjestelmissä vähemmän luotettavaa.

Generoidut ohjelmakoodit voidaan myös tallentaa versionhallintaan, jolloin ne pysyvät siellä heti käyttöä tai tarkastelua varten. Koodiskannaustyökalujen avulla voidaan myös esimerkiksi tarkastaa, sisältääkö generoitu ohjelmakoodi haavoittuvuuksia. Ohjelmakoodi voidaan myös pitää poissa versionhallinnasta, jolloin kehittäjät generoivat koodin ainoastaan omalla tietokoneella ja käyttävät sitä ohjelmistoriippuvuutena ohjelmiston ajamiseksi. Tästä voi olla hyötyä eri versioiden välillä, mutta monesti halutaan ylläpitää vain tiettyä versiota. Organisaatiot voivat siis itse päättää, mikä on sopivin menettelytapa kehitystiimille.

Ohjelmakoodin generoinnista eri ohjelmointikielille on hyötyä etenkin mikropalveluissa, koska mikropalvelut voidaan kehittää eri ohjelmointikielillä. Generoidut koodit voivat olla eri ohjelmointikielillä asiakaspuolella ja palvelinpuolella. Palvelut voidaan määritellä yhteisellä rajapinnan määritelmäkielellä, joka on riippumaton yksittäisistä teknologioista. Tämän jälkeen ohjelmakoodit voidaan generoida mille tahansa ohjelmointikielelle, jolle on olemassa gRPC-kirjasto. (Babal 2023, luku 1.1.2.) gRPC:tä ei siis ole lukittu mihinkään tiettyyn teknologiaan. gRPC:llä voidaan generoida

asiakaspuolen koodi esimerkiksi Java-ohjelmointikielelle ja palvelinpuolen koodi Go-ohjelmointikielelle. Javan puolelta voidaan sitten kutsua Go-puolen etäproseduurikutsuja.

4.2.3 Datan suoratoisto

Joissain tapauksissa on hyödyllistä palauttaa pyyntöjen vastaukset takaisin asiakasohjelmalle pienissä osissa nopeasti, jolloin kaikkea dataa ei palauteta kerralla. Asiakasohjelmat tarvitsevat välillä vain tietyn osan vastauksesta kerralla, joten aina ei ole järkevää palauttaa kaikkea dataa yhdessä isossa osassa. (Babal 2023, luku 1.1.5.) gRPC:n avulla tämä on mahdollista. gRPC:n suoratoistotilassa voidaan lähettää tai palauttaa dataa osissa, mikä vähentää tiedonsiirron viivettä, jos data on liian iso lähetettäväksi tai palautettavaksi kerralla. Tämän ansiosta asiakasohjelmat voivat vastaanottaa dataa jatkuvasti ilman, että niiden tarvitsee odottaa koko vastauksen saapumista. Tämä on hyödyllinen gRPC:n ominaisuus etenkin reaaliaikasovelluksissa, kuten chateissa tai verkkovideopeleissä. (Kong 13.6.2024.)

Kun esimerkiksi pyynnön lähettävä mikropalvelu saa tietyn osan vastauksesta, se voi suorittaa sen avulla jonkin tietyn toiminnon ja lähettää pyynnön toiselle mikropalvelulle ennen kuin kaikki data on palautettu. Tämä nopeuttaa palveluiden toimintaa. Ensimmäisessä datan osassa saattaa olla kaikki ne tiedot, joita palvelu tarvitsee jonkin tietyn toiminnon suorittamiseen. Tästä syystä suoratoistosta voi olla hyötyä joissain tapauksissa. Tyypillisessä tapauksessa yhteys avataan asiakkaan ja palvelimen välille kerran, jonka jälkeen kaikki data siirretään tämän avatun yhteyden kautta (Babal 2023, luku 1.1.5). Ei siis tarvitse avata uusia yhteyksiä jokaiselle siirrettävälle osalle. Tämä on mahdollista HTTP/2-protokollan ansiosta. Kuten luvussa 4.2.1 mainittiin, sen avulla voidaan lähettää useita pyyntöjä samalla yhteydellä. On hyvä kuitenkin huomata, että datan suoratoisto toimii asynkronisesti. Sitä ei ole tarkoitettu synkroniseen kommunikointiin.

4.3 gRPC:n haitat

Vaikka gRPC tarjoaa monia hyötyjä, sillä on olemassa myös haittoja, joista kannattaa olla tietoinen. Ensinnäkin gRPC API ei ole yhtä yksinkertainen ja helppo toteuttaa kuin esimerkiksi perinteinen REST API. gRPC vaatii yleensä Protocol Buffers -kääntäjän, jolla saadaan käännettyä prototiedostoihin määritetyt gRPC-palvelut ohjelmointikielien tietorakenteisiin (Google 2024). gRPC:n käyttämä Protocol Buffers saattaa olla monelle kehittäjälle asteen haastavampi oppia ja käyttää kuin yleiset tekstipohjaiset sarjoittamismuodot, kuten JSON tai XML (The Postman Team 13.11.2023). Tämä voi viedä aikaa kehityksessä joissakin tiimeissä.

gRPC vaatii prototiedostojen läsnäolon, jotta sitä voidaan testata. gRPC-pyyntöjä ei kirjoiteta käsin binäärimuodossa, vaan pyynnöt tehdään prototiedostojen avulla. Tästä syystä testaaminen ei ole yhtä yksinkertaista kuin pelkän HTTP-pyynnön lähettäminen JSON-datalla. (Broshar 27.4.2023.)

gRPC vaatii erillisesti luotavan asiakasohjelman, joka on tietoinen gRPC-palvelimen etäproseduurikutsuista. gRPC API:a ei siis voida testata suoraan esimerkiksi curl-ohjelmalla komentoriviltä toisin kuin REST API:a (Broshar 27.4.2023). Tähän on kuitenkin olemassa ratkaisu, johon käytetään gRPC-palvelimen tarjoamaa heijastustoimintoa. Tästä kerrotaan tarkemmin luvussa 5.7.

Yksi merkittävä haittapuoli gRPC:ssä on se, että verkkoselaimet eivät tällä hetkellä tue sitä synnynnäisesti. Koska gRPC käyttää matalan tason pääsyä HTTP/2-protokollan ominaisuuksiin, verkkoselaimet eivät pysty tekemään tarvittavia verkkopyyntöjä gRPC-asiakkaan tukemiseksi. Tähän on kuitenkin olemassa teknologia nimeltä gRPC-Web. Sen avulla voidaan yhdistää verkkosovellukset selaimista gRPC-palveluihin välityspalvelimen kautta. Se ei kuitenkaan ole täysin yhteensopiva gRPC:n kanssa, minkä takia sen käyttö on vähäistä. (Xu & Lam 1.12.2022, 4:45–5:30 min.) Verkkoselainten heikko tuki ei kuitenkaan ole ongelma mikropalveluiden välisessä kommunikoinnissa, koska mikropalvelut ajetaan palvelimilla.

gRPC on myös melko uusi teknologia, joten kaikki kolmannen osapuolen myyjät eivät välttämättä tue sitä vielä kovin hyvin (Kong 27.4.2022). Kolmannet osapuolet voivat olla esimerkiksi pilvipalveluiden tarjoajia. Lisäksi gRPC:n työkalujen määrä voi olla vielä vähäistä, mutta koska projekti on avointa lähdekoodia, sille voidaan odottaa tulevan uusia työkaluja tulevaisuudessa. Vaikka gRPC on melko uusi, se on kuitenkin saanut paljon suosioita sen julkaisun jälkeen. Tällä hetkellä siitä löytyy paljon dokumentaatioita ja ohjeita internetistä, sekä siihen liittyen on myös kirjoitettu kirjoja.

Vaikka gRPC:hen liittyy muutamia haittapuolia, sen tuomat hyödyt ovat kuitenkin haittoja suuremmat (Kong 27.4.2022). gRPC:tä ei siis kannata sulkea pois sen haittojen takia, koska se antaa paljon hyötyjä pitkällä aikavälillä. gRPC saattaa olla aluksi haastava oppia etenkin kokemattomille ohjelmistokehittäjille, mutta oikein toteutettuna se parantaa palveluiden kommunikointia paljon.

4.4 gRPC-protokollan soveltuvuus

gRPC on erinomainen teknologia. Sen pääasiallinen käyttötarkoitus on kuitenkin hyvä tietää, koska se ei sovi kaikkiin sovelluksiin. gRPC:n pääasiallinen käyttötarkoitus on olla kommunikointimekanismi mikropalveluiden välillä datakeskuksissa (Xu & Lam 1.12.2022, 5:20–5:45). gRPC:tä voidaan hyödyntää myös esimerkiksi mobiilisovelluksien ja palvelimien välisessä kommunikoinnissa, mutta sitä ei ole luotu siihen.

Mikropalveluiden maailmassa iso osa kommunikoinnista tapahtuu yleensä asynkronisilla kommunikointitavoilla. Jotkin toiminnot kuitenkin vaativat suoraa kommunikointia mikropalveluiden välillä. gRPC tulisi olla pääasiallinen valinta mikropalveluiden synkroniselle kommunikoinnille, jossa mikropalvelut kommunikoivat suoraan keskenään ilman viestinvälittäjiä. Sen erinomainen suorituskyky, joka perustuu HTTP/2-protokollaan ja Protocol Buffersiin, tekee siitä täydellisen valinnan. (Ozkaya

7.9.2021.) Jos nykyään kehitetään uusia mikropalveluita ja niiden täytyy kommunikoida suoraan keskenään tehokkaasti, kannattaa siis valita gRPC.

gRPC sopii hyvin mikropalvelusovelluksiin, joissa yksittäisiä palveluita kehitetään eri ohjelmointikielillä, ja joissa palvelut saavat vaihtelevan määrän kuormitusta (The Postman Team 20.11.2023). gRPC:n hyvän suorituskyvyn avulla palvelut voivat siis käsitellä järjestelmään tulevaa kuormitusta tehokkaammin. Tämän takia sitä kannattaa käyttää palveluissa, joihin tulee eniten verkkopyyntöjä, kuten sovelluksen toimivuuden kannalta kriittisimmissä ja tärkeimmissä palveluissa. Toki kaikkein eniten hyödytään siitä, jos kaikki mikropalveluiden välinen synkroninen kommunikointi tehdään gRPC:llä. Ei ole kauheasti järkeä rakentaa muutaman mikropalvelun API:a esimerkiksi REST:illä, jos kaikki muut käyttävät gRPC:tä. Tässä tilanteessa tulisi vain turhaan väistettyä gRPC:n hyötyjä. Ehkä keskeisimpiä syitä sille, miksi mikropalveluiden synkronista kommunikointia ei toteutettaisi gRPC:llä, on kehittäjien tietämättömyys gRPC:n hyödyistä, sekä osaamisen puute gRPC:n käyttämisestä ja toteuttamisesta.

Tällä hetkellä työkalujen tuki gRPC:lle selainohjelmien puolella on vähäistä. Siksi on luontaista käyttää gRPC:tä ympäristöissä, jotka tukevat sitä synnynnäisesti, kuten palvelimilla. On kuitenkin olemassa työkaluja, joiden avulla voidaan luoda gRPC API:sta HTTP API. Yksi esimerkki on gRPC Gateway, jonka avulla voidaan kääntää HTTP-pyyntöjä gRPC-pyynnöiksi välityspalvelimen kautta (Broshar 27.4.2023). gRPC Gatewayn avulla voidaan luoda gRPC-palvelusta REST API, jonka kautta pyynnöt kulkevat gRPC-palvelusta ulos ja sisään. Tämä on hyödyllistä, jos asiakasohjelmat haluavat kommunikoida gRPC-palvelun kanssa helpommin käytettävällä REST:illä, tai jos niillä on heikko tuki gRPC:lle.

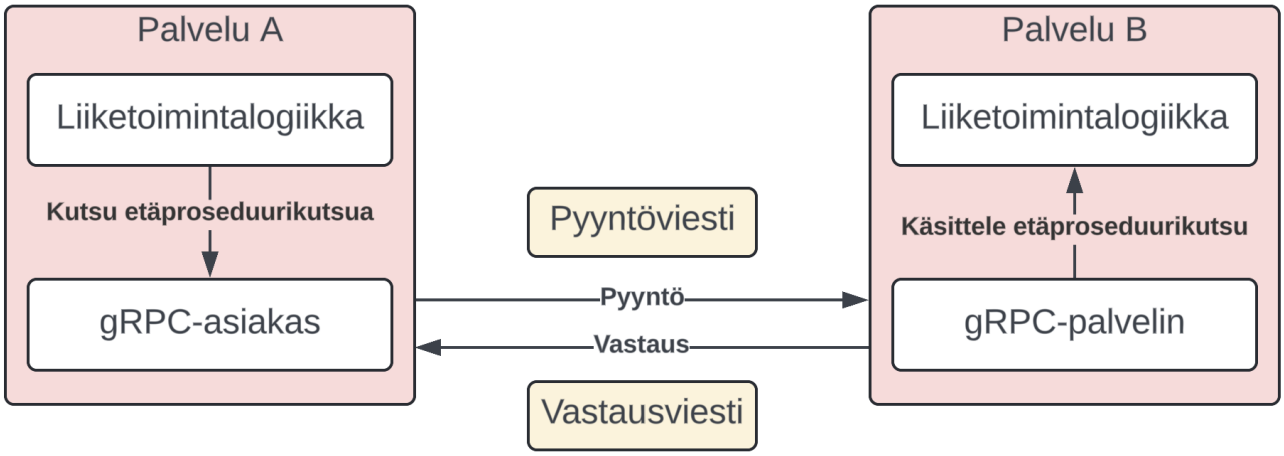
4.5 gRPC-protokollan viestintämallit

gRPC:n kommunikointi pohjautuu pyyntöihin ja vastauksiin. Se käyttää pääasiallisesti synkronista kommunikointitapaa, mutta voi kuitenkin toimia myös täysin asynkronisesti suoratoistotilassa, kun asiakkaan ja palvelimen välinen yhteys on luotu. (Indrasiri & Kuruppu 2020, luku 1.) gRPC:n avulla voidaan hyödyntää erityyppisiä etäproseduurikutsuja kommunikoinnissa yksinkertaisen pyyntö-vastaustyylin lisäksi (Indrasiri & Kuruppu 2020, luku 3). gRPC:n viestintämallit ovat

- yksinkertainen etäproseduurikutsu
- palvelinpuolen suoratoisto
- asiakaspuolen suoratoisto
- kaksipuoleinen suoratoisto.

4.5.1 Synkroninen viestintämalli

Yksinkertainen etäproseduurikutsu on gRPC:n yksinkertaisin viestintämalli. Siinä gRPC-asiakas lähettää gRPC-palvelimelle yhden pyyntöviestin etäproseduurikutsun yhteydessä ja saa vastauksena takaisin yhden vastausviestin. Vastauksen yhteydessä lähetetään tietoja palvelimen tilasta vastauksen mukana tulevan metadatan lisäksi. (Indrasiri & Kuruppu 2020, luku 3 alaluku Simple RPC (Unary RPC).) Tämä etäproseduurikutsu on samantyyppinen kuin perinteinen HTTP-pyyntö (The Postman Team 13.11.2023). Yksinkertaista etäproseduurikutsua voidaan siis pitää perinteisenä synkronisena asiakas-palvelinkommunikointina. Kun gRPC:tä käytetään mikropalveluiden synkronisessa kommunikoinnissa, käytetään tätä viestintämallia. Kuvassa 11 näkyy, miten gRPC:n yksinkertainen etäproseduurikutsu toimii.



Kuva 11. Yksinkertaisen etäproseduurikutsun toiminta (mukaillen Indrasiri & Kuruppu 2020, luku 3 alaluku Simple RPC (Unary RPC))

Pyyntöjen ja vastausten viestit sisältävät dataa, jota halutaan siirtää prosessien välillä. Pyyntöviesti voi siis sisältää dataa liittyen esimerkiksi suoritettavaan toimintoon, jota gRPC-palvelimen puolella tarvitaan pyynnön käsittelyksi liiketoimintalogiikassa. Vastausviesti taas voi sisältää dataa liittyen suoritettavan toiminnon tulokseen. Jos halutaan esimerkiksi hakea tuotetiedot, niin pyyntöviesti voi sisältää haettavan tuotteen ID:n ja vastausviesti sisältää itse tuotteen tiedot.

Yksinkertainen etäproseduurikutsu on gRPC:n viestintämalleista helpoin toteuttaa. Sen ohjelmoiminen on hyvin suoraviivaista verrattuna muihin etäproseduurikutsutyyppeihin, koska tarvitsee miettiä vain yhtä pyyntöä ja vastausta, jotka käsitellään synkronisesti. Tämän opinnäytetyön osana toteutettavassa projektissa käytetään tätä etäproseduurikutsutyyppiä.

4.5.2 Asynkroniset viestintämallit

Ensimmäinen gRPC:n asynkronisista viestintämalleista on palvelinpuolen suoratoisto. Siinä gRPC-asiakas lähettää gRPC-palvelimelle pyyntöviestin ja saa vastauksena sarjan vastausviestejä. Tätä viestien sarjaa kutsutaan suoratoistoksi. Kun kaikki vastausviestit ovat lähetetty, gRPC-palvelin merkitsee suoratoiston päättyneeksi lähettämällä gRPC-asiakkaalle tietoja palvelimen tilasta. (Indrasiri & Kuruppu 2020, luku 3 alaluku Server-Streaming RPC.) Tämä viestintämalli on hyödyllinen, kun palvelimen täytyy käsitellä hyvin paljon dataa (Kong 27.4.2022).

Toinen asynkroninen viestintämalli on asiakaspuolen suoratoisto. Siinä gRPC-asiakas lähettää gRPC-palvelimelle useamman pyyntöviestin ja vastaanottaa yhden vastausviestin. Palvelimen ei kuitenkaan tarvitse odottaa, että se saa asiakkaalta kaikki viestit ennen kuin se lähettää vastauksen. Se voi lähettää vastauksen parin ensimmäisen viestin jälkeen, tai kun kaikki viestit ovat lähetetty. (Indrasiri & Kuruppu 2020, luku 3 alaluku Client-Streaming RPC.) Tämä etäproseduurityyppi sopii hyvin tilanteisiin, joissa verkkokutsuista aiheutuva viive on huolenaiheena (Kong 27.4.2022).

Kolmas asynkroninen viestintämalli on kaksipuoleinen suoratoisto. Se on ikään kuin kahden edellisen viestintämallin yhdistelmä. Kaksipuoleisessa suoratoistossa gRPC-asiakas lähettää gRPC-palvelimelle pyynnön, joka muodostuu useammasta viestistä. Palvelin käsittelee nämä viestit ja lähettää vastauksena sarjan viestejä. Tämän tyyppinen etäproseduurikutsu pitää aloittaa gRPC-asiakkaan puolelta, mutta sen jälkeen kommunikointi tapahtuu asiakkaan ja palvelimen ohjelmalogiikan mukaan. (Indrasiri & Kuruppu 2020, luku 3 alaluku Bidirectional-Streaming RPC.)

Kaksipuoleinen suoratoisto on gRPC:n monimutkaisin viestintämalli ja se saattaa olla haastava toteuttaa. Sitä kuitenkin harvoin käytetään yksinkertaisissa käyttötarkoituksissa, koska se on tarkoitettu edistyneempiin asynkronisiin kommunikointitarpeisiin. Mikropalveluiden välisessä kommunikoinnissa sitä ei yleensä käytetä, jos kommunikointi halutaan toteuttaa synkronisesti. Se on erinomainen valinta esimerkiksi reaaliaikaiseen kommunikointiin (Kong 27.4.2022).

4.6 gRPC:n tekninen toiminnallisuus

gRPC on monipuolinen teknologia. Sen olennaiset osat kannattaa tiedostaa hyvin, jotta pystyy vertailemaan paremmin sen soveltuvuutta omiin tarpeisiin. gRPC:n käyttäminen mikropalveluiden kommunikoinnissa voi olla suuri investointi, joten on tärkeää olla tietoinen sen käytöstä. Seuraavaksi käydään läpi tarkemmin, miten gRPC toimii teknillisesti.

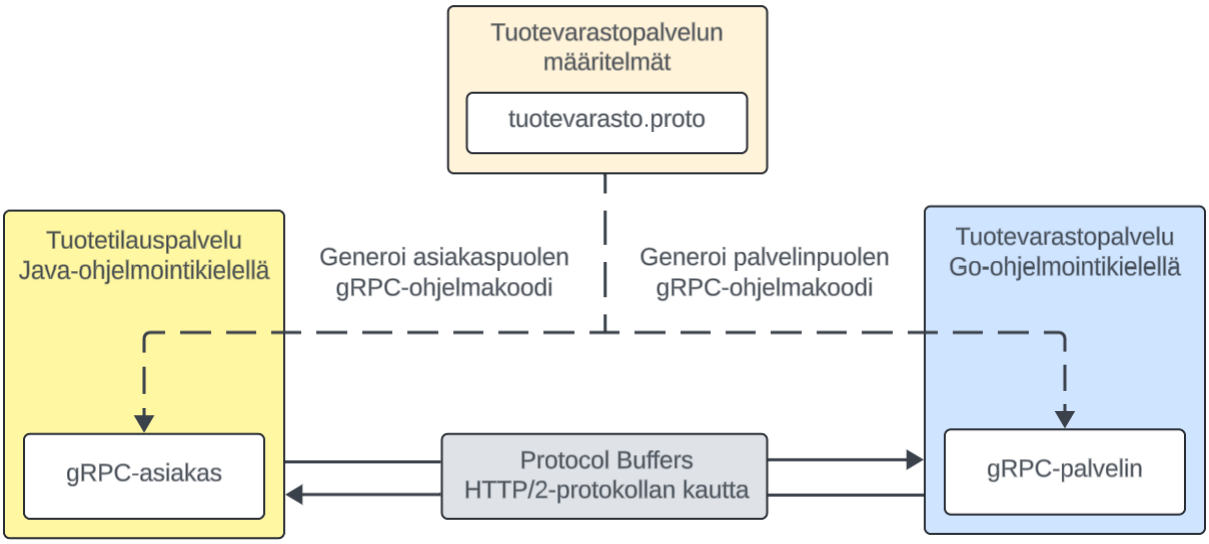
4.6.1 Palveluiden määritteleminen Protocol Buffersia käyttäen

Kun gRPC:tä aletaan toteuttamaan mikropalvelulle, ensimmäisenä määritellään sen rajapinta. Palvelun rajapinta sisältää tietoa siitä, miten palvelua käyttävät asiakasohjelmat voivat käyttää sitä. Kieli, jota palvelun rajapinnan määrittelyssä käytetään, kutsutaan rajapinnan määritelmäkieleksi (englanniksi interface definition language). (Indrasiri & Kuruppu 2020, luku 1 alaluku What is gRPC?) gRPC:ssä määritelmät tehdään useimmiten Protocol Buffersilla. Protocol Buffersilla määritelmät kirjoitetaan tavallisiin tekstitiedostoihin, joilla on proto-tiedostopääte (Indrasiri & Kuruppu 2020, luku 1 alaluku Service Definition).

Protocol Buffers tai lyhyemmin protobuf on datan sarjoittamiseen tarkoitettu mekanismi. Oletuksena gRPC käyttää Protocol Buffersia sen rajapinnan määritelmäkielenä ja datan sarjoittamismuotona. (Borysov & Gardiner 3.9.2021.) Protocol Buffers on ohjelmointikieli- ja alustaneutraali, joten sitä voidaan käyttää millä tahansa ohjelmointikielellä ja laitteella. Se on kuin JSON jäsennetyn datan sarjoittamisessa, mutta tehokkaampi. Protocol Buffersin on kehittänyt Google ja se on avointa lähdekoodia. (Google 2024.)

Prototiedostoissa on määriteltynä viestejä ja palveluiden funktioita. Viestit voivat olla esimerkiksi pyyntö- ja vastausobjekteja jollekin tietylle mikropalvelulle, joita se käyttää funktioidensa parametreissa ja paluuarvoina. (Babal 2023, luku 2.5.1.) Viestit sisältävät kenttiä, joilla voi olla eri tietotyyppejä ja arvoja. Kaikki viestin kentät sisältävät uniikin numeron, joka yksilöllistää sen viestin tietorakenteessa. Viestit voidaan määritellä itse, mutta Protocol Buffers sisältää myös valmiiksi määriteltyjä viestejä. Määriteltävät palvelut sisältävät funktioita, joita voidaan kutsua etänä. Kun kaikki määritelmät on tehty tiedostoon, siitä voidaan generoida asiakas- ja palvelinpuolen ohjelmakoodit gRPC:tä käyttäville eri ohjelmointikielillä toteutetuille mikropalveluille. (Indrasiri & Kuruppu 2020, luku 1 alaluku Service Definition.) Ohjelmakoodit generoidaan käyttäen Protocol Buffersin kääntäjää. Kääntäjä on tietokoneohjelma, jonka täytyy olla olemassa ohjelmakoodia generoivalla tietokoneella.

Generoidut ohjelmakoodit sisältävät abstraktioita matalan tason kommunikoinnin piilottamiseksi. Se helpottaa ja yksinkertaistaa gRPC:llä toteutettujen API:en ohjelmointia, koska matalan tason kommunikointilogiikka, jonka avulla palvelut voivat kommunikoida keskenään, piilotetaan koodigeneroinnin yhteydessä. gRPC hoitaa kaikki kommunikointiin liittyvät monimutkaisuudet, jolloin kehittäjät voivat keskittyä enemmän ohjelmalogiikkaan. (Indrasiri & Kuruppu 2020, luku 1 alaluku What is gRPC?) Kuva 12 havainnollistaa, miten gRPC toimii kokonaisuudessaan kahden mikropalvelun välillä.

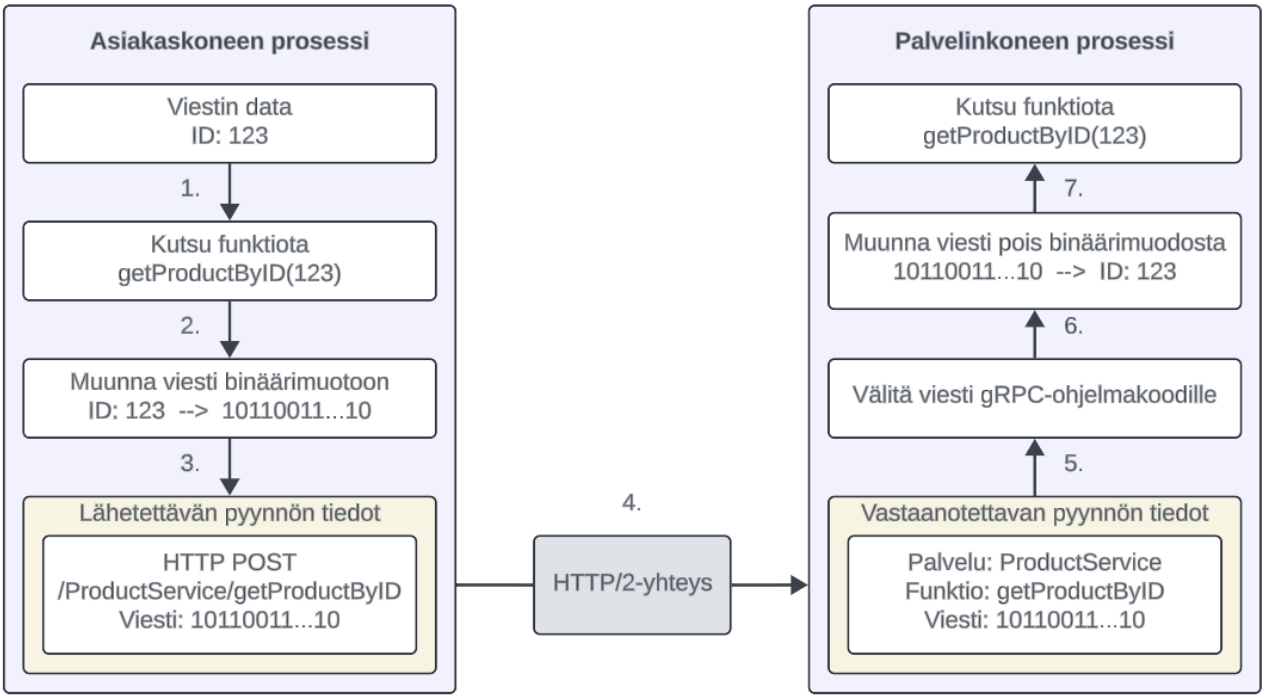


Kuva 12. gRPC:n käyttö kahden mikropalvelun välillä (mukaillen Indrasiri & Kuruppu 2020, luku 1)

4.6.2 Etäproseduurikutsun kulku

Mikropalveluissa asiakkaan ja palvelimen välinen kommunikointi tarkoittaa palveluiden välistä kommunikointia. Asiakkaalla viitataan toista palvelua kutsuvaan palveluun ja palvelimella viitataan kutsuttavaan palveluun. gRPC-asiakas voi kutsua gRPC-palvelinta generoidulla gRPC:n asiakaspuolen ohjelmakoodilla. (Babal 2023, luku 2.5.3.)

Kun gRPC-asiakas kutsuu gRPC-palvelinta, asiakaspuolen gRPC-ohjelmakoodi muuntaa etäproseduurikutsun tietorakenteet binäärimuotoon, jossa kutsu lähetetään. Palvelinpuolen gRPC-ohjelmakoodi hoitaa binäärimuodon muuntamisen takaisin gRPC-ohjelmakoodin tietorakenteisiin. Kommunikointi tapahtuu käyttäen HTTP/2-protokollaa, jonka avulla data voidaan lähettää binäärimuodossa. (Indrasiri & Kuruppu 2020, luku 1 alaluku Client-Server Message Flow.) gRPC-protokollassa kaikki verkkopyynnöt tehdään HTTP-protokollan POST-metodilla. Kutsuttava funktio ja palvelu, jossa funktio on, lähetetään erillisessä HTTP-otsikossa. Kun pyyntö tulee palvelimelle, palvelin tarkistaa pyynnön tiedoista kutsuttavan funktion. Tämän jälkeen pyynnön mukana tullut viesti jäsennetään sen ohjelmointikielen tietorakenteisiin, jolla gRPC-palvelin on ohjelmoitu. Jäsennetyllä viestillä voidaan kutsua palvelun funktiota antamalla viesti funktiolle parametrina. (Indrasiri & Kuruppu 2020, luku 4 alaluku RPC Flow.) Kuvassa 13 on kuvattuna tämä prosessi, miten gRPC:llä toteutettu etäproseduurikutsu kulkee verkon yli.



Kuva 13. Etäproseduurikutsun kulku verkon yli gRPC-protokollaa käyttäen (mukaillen Indrasiri & Kuruppu 2020, luku 4)

4.6.3 Etäproseduurikutsun metadata ja aikaraja

Pyyntöjen ja vastausten mukana kulkee yleensä myös metadataa, joka pitää sisällään tietoa etäproseduurikutsusta. Siihen voidaan liittää esimerkiksi autentikointitietoja, joiden avulla palvelut voivat todentaa itsensä muille palveluille tietoturvan parantamiseksi. Lisäksi kutsujen mukana kulkee etäproseduurikutsun aikaraja, joka määrittelee gRPC-palvelimella maksimiaikarajan, jonka jälkeen kutsu perutaan. Nämä molemmat ovat ohjelmointikielestä riippuvia. (gRPC Authors 2022.) Metadata ja aikaraja pitää siis itse määritellä gRPC-asiakkaan ja gRPC-palvelimen ohjelmakoodin logiikassa. Ne ovat erilaisia eri ohjelmointikielillä, mutta ovat tärkeitä tuotannollisessa gRPC-kommunikoinnissa.

Metadatan avulla voidaan lähettää dataa ilman, että sitä tarvitsee määritellä prototiedostoihin. Tämä on hyödyllistä, kun etäproseduurikutsuja on paljon ja samaa dataa halutaan lähettää kaikkien etäproseduurikutsujen yhteydessä. Ilman aikarajaa taas etäproseduurikutsut jäisivät odottamaan vastauksen saantia ikuisesti, jos palvelimella tapahtuu jokin vika. Joidenkin ohjelmointikielien gRPC-toteutuksissa ei välttämättä ole oletusaikarajaa (gRPC Authors 2022). Tästä syystä se olisi aina hyvä asettaa. Aikaraja ei kuitenkaan kannata olla liian lyhyt tai liian pitkä, vaan sen tulisi olla sopiva etäproseduurikutsun mukaan.

Etäproseduurikutsut gRPC:ssä pitävät siis sisällään muutakin kuin pelkkiä pyyntö- ja vastausviestejä. Niissä on useita yksityiskohtia ja ominaisuuksia, joiden toteutus eroaa eri ohjelmointikielillä. Tämän opinnäytetyön osana toteutettavassa projektissa ei käytetä kaikkia näitä gRPC:n ominaisuuksia, vaan siinä pyritään enemmän yleisen kuvan saamiseen gRPC:stä.

5 Esimerkkinä toteutetun mikropalveluprojektin kehityksen kuvaus

Tässä luvussa kuvataan tämän opinnäytetyön osana toteutettu ohjelmistoprojekti, jossa kehitetään kolme mikropalvelua, jotka kommunikoivat synkronisesti keskenään käyttäen gRPC:tä. Tuottaminen keskittyy gRPC:n tekniseen toteutukseen ja käyttöön. Projekti on melko laaja, joten sen kaikkia osia ei käydä läpi yksityiskohtaisesti. Jotkin kohdat tiivistetään ja jätetään käsittelemättä toiston vähentämiseksi.

Projekti on julkaistu avoimena lähdekoodina GitHub-verkkopalvelussa. Tämä tarkoittaa sitä, että kuka tahansa voi käyttää ja tarkastella tuotosta. Mikropalveluiden lähdekoodit ovat tarkasteltavissa osoitteessa https://github.com/hollowdll/go-grpc-microservices. gRPC API:en määritelmät sisältävät prototiedostot ja niistä generoidut gRPC-ohjelmakoodit Go-ohjelmointikielelle löytyvät osoitteesta https://github.com/hollowdll/grpc-microservices-proto.

5.1 Projektin lähtökohta

Toteutettava projekti on kolmesta mikropalvelusta koostuva pieni kokonaisuus. Ensimmäinen palvelu käsittelee tuotteiden tilauksia, toinen palvelu käsittelee tuotevarastoa ja kolmas palvelu käsittelee maksuja. Palvelut muodostavat toiminnon, joka havainnollistaa verkkokaupassa tehtäviä tuotteiden tilauksia. Toiminto on gRPC:llä toteutettava synkroninen kommunikointiketju, joka jakautuu näihin kolmeen mikropalveluun.

Mikropalvelut yksinkertaistetaan tässä projektissa, eikä niitä suunnitella kokonaiseen sovellukseen. Tästä syystä ne eivät kuvaa todellista mikropalvelusovellusta. Kehitettävä tuotevarastopalvelu tiivistetään käsittelemään sekä tuotteita, että niiden varastomäärää. Oikeassa järjestelmässä nämä saatetaan jakaa kahteen eri palveluun paremman eristyksen vuoksi, mikä on mikropalveluiden ydintarkoituksia. Kehitettävä maksupalvelu ei käsittele oikeaa rahaa, eikä eri maksutapoja, koska tuotos ei tule oikeaan käyttöön. Siinä on tarkoitus simuloida maksutapahtumia.

Tuotos kehitetään ilmaisilla ja avoimen lähdekoodin työkaluilla. Tästä syystä projektin tuottaminen työkalujen osalta ei tuo kustannuksia. Projektissa käytettävät kehittämismenetelmät valitaan niiden sopivuuden kannalta. Koska projektia kehitetään yksin, eikä työllä ole toimeksiantajaa, tuottamisella on hyvin vähän rajoitteita. Projektilla on kuitenkin tavoitteet ja vaatimukset, joiden mukaan kehitystä tehdään.

Mikropalvelut ohjelmoidaan Go-ohjelmointikielellä. Go on Googlen luoma avoimen lähdekoodin ohjelmointikieli, jota käyttää nykyään monet maailmanlaajuiset yritykset ja organisaatiot eri aloilla ohjelmistoissaan ja palveluissaan (Google s.a.). Se sopii erinomaisesti mikropalveluiden

rakentamiseen ja sillä on erittäin hyvä tuki gRPC:lle. Valitsin Go-kielen käytettäväksi ohjelmointikieleksi, koska halusin saada kyseisen kielen kanssa työskentelystä lisää kokemusta.

Tuotosta voidaan pitää onnistuneena, jos mikropalvelut saadaan toteutettua minimaalisiksi siten, että ne voivat kommunikoida synkronisesti keskenään gRPC:n avulla suunniteltavan tuotetilaustoiminnon mukaan. Kommunikoinnin toimivuus voidaan testata manuaalisesti lähettämällä eri pyyntöjä jokaiseen mikropalveluun. Lopullinen tuotetilaustoiminto voidaan testata ajamalla kaikki mikropalvelut, jonka jälkeen tilauspalveluun voidaan tehdä onnistuneita tilauspyyntöjä. Tilauspyyntö palauttaa vastausdatan tai virhekoodin vian sattuessa. Mikropalveluiden lokeja seuraamalla saadaan tietää, että pyynnöt tulevat perille.

Tämän projektin mikropalveluita ei suunnitella tuotannollisiksi, eivätkä ne sisällä kaikkia asioita, joita tuotannolliset mikropalvelut tulisivat sisältää. Esimerkiksi autentikointi ja salaus jätetään tässä projektissa syrjään, koska ne eivät ole projektin onnistumisen kannalta olennaisia. Tietokantoihin ei myöskään keskitytä, koska ne ovat tämän projektin rajauksen ulkopuolella. Lisäksi mikropalveluiden välillä siirrettävä data ei ole tässä projektissa olennaista. Siirrettävä data on havainnollistavaa esimerkkidataa, jotta datan siirtäminen gRPC:n avulla voidaan näyttää.

5.2 Projektin hyödynnettävyys

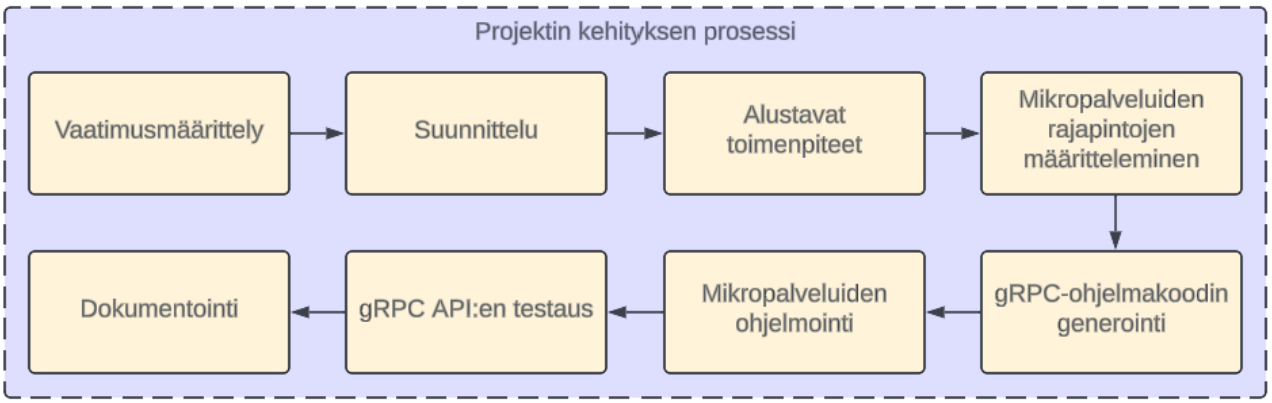
Tuotoksella ei ole toimeksiantajaa, joten se ei tule oikeaan liiketoimintakäyttöön. Tuotoksen on tarkoitus olla havainnollistava esimerkkiprojekti mikropalveluiden toteutuksesta, jossa palveluiden kommunikointi toteutetaan synkronisesti gRPC:llä. Sen pääasiallisimpia tarkoituksia onkin olla osana omaa ja muiden oppimista. Jatkossa sitä voidaan hyödyntää esimerkiksi tulevissa projekteissa tai hyödyllisenä materiaalina. Sitä voidaan hyödyntää materiaalina esimerkiksi tilanteissa, joissa mikropalveluiden API:t halutaan korvata gRPC:llä, tai jos halutaan kehittää sovelluksia, jotka käyttävät synnynnäisesti gRPC:tä. Projektista myös ilmenee, miten palveluiden rajapinnat voidaan toteuttaa käyttäen Protocol Buffersia, ja miten Protocol Buffersin kanssa voidaan työskennellä.

Tuotoksen kohderyhmänä on ensisijaisesti ohjelmistokehittäjät. Vaikka tuotos kohdentuukin enemmän ohjelmistokehittäjille, joilla on jo hieman tietämystä mikropalveluista, sitä voi kuitenkin hyödyntää myös muut ohjelmistokehittäjät. gRPC:tä ei käytetä pelkästään mikropalveluissa, joten työstä saadaan myös hyvä kuva gRPC:n teknisestä toteutuksesta ja käytöstä yleisesti.

Koska työn hyödynnettävyys on olennainen osa projektia, työn lopputuloksen on tarkoitus vastata siihen. Tästä syystä tuotoksen kehityksessä hyödynnetään ammatillisia ohjelmistokehitysmenetelmiä ja parhaita käytänteitä. Lisäksi projektin lähdekoodi ja dokumentaatiot kirjoitetaan englanniksi laajemman kohdeyleisön saavuttamiseksi. Työn kohderyhmä on hyvin kansainvälinen, joten sitä ei ole lukitu ainoastaan suomalaisiin.

5.3 Projektin vaatimusmäärittely ja suunnittelu

Tuottaminen koostui eri työvaiheista, joiden mukaan projektia kehitettiin järjestyksessä. Järjestystä ei kuitenkaan lukittu, ja palasin myöhemmin osaan vaiheista toisten vaiheiden jälkeen. Esimerkiksi suunnitteluun palasin vielä hetkeksi ennen mikropalveluiden ohjelmointia ja toteutettujen gRPC API:en testausta tein rinnakkain mikropalveluiden ohjelmoinnin kanssa. Kokonaisuudessaan projektin työvaiheet kuitenkin etenivät kuvan 14 mukaisesti.



Kuva 14. Projektin kehityksen prosessi

Kehitys lähti liikkeelle suunnittelusta, jota edelsi ensin kevyt vaatimusmäärittely. Vaatimusmäärittelyssä mietin ja päätin, mitä kaikkea projektiin tulee. Luvussa 1 mainittiin projektin minimivaatimukset. Toimeksiantajaa projektissa ei ollut, joten määrittelin projektin minimivaatimukset kohderyhmää ajatellen. Vaatimuksena oli, että mikropalvelut kehitetään minimaalisiksi siten, että ne voivat kommunikoida synkronisesti gRPC:llä suunniteltavan kommunikointiketjun mukaisesti. Projektissa oli tarkoituksena esitellä tarvittavat toimenpiteet, joiden avulla synkroninen gRPC-kommunikointi saadaan toteutettua mikropalveluille. Tämän avulla kohderyhmä saisi hyvän käsityksen, mitä kaikkea gRPC:n toteuttamisessa tulee ottaa huomioon.

Käytin suunnittelussa ideointimenetelmänä ajatuskarttaa projektin sisältöjen visualisoimiseksi. Ajatuskartan piirsin avoimen lähdekoodin kuvanpiirto-ohjelmistolla nimeltä Excalidraw. Piirsin keskelle projektin, joka jakoi sen sisällöt omiin haaroihin (liite 2). Tämä auttoi hahmottamaan, mitä kaikkea projektissa otetaan huomioon, ja mitä siihen tulee. Suunnittelun edetessä tein toisen version alkuperäisestä ajatuskartasta, johon tuli päivitetty suunnitelma ennen varsinaisen toteutuksen aloittamista. Alkuperäinen suunnitelma oli aika laaja, joten karsin siitä kohtia pois, jotka eivät olleet projektin onnistumisen kannalta olennaisia. Esimerkiksi ulkoiset tietokannat ja API-yhdyskäytävän päätin vasta suunnittelun jälkeen ottaa pois, vaikka ne olivat alun perin tarkoitus sisällyttää projektiin.

Kun minulla oli tiedossa, mitä projekti tuli sisältämään, aloitin suunnittelemaan sen rakennetta ja toiminnallisuutta tarkemmin. Tässä kohtaa hyödynsin erilaisia kaavioita hahmottamisen helpottamiseksi. Piirsin ensin arkkitehtuurikaavion kehitettävän projektin rakenteesta. Se auttoi hahmottamaan teknistä toteutusta visualisoimalla projektin komponentteja ja niiden yhdistämistä. Liitteestä 3 näkyy suunnittelun aikana syntynyt projektin arkkitehtuurikaavio. Tämä arkkitehtuurikaavio kuitenkin poikkesi paljon projektin lopullisesta tuloksesta. Arkkitehtuurikaavion jälkeen piirsin projektille sekvenssikaavion. Se kuvaa projektin sisältämän tuotetilaustoiminnon kommunikointiketjun kulkua (liite 4). Mikropalvelut tulivat kommunikoimaan tämän kaavion mukaisesti. Sekvenssikaavio auttoi ymmärtämään paremmin, mitä kehityksen aikana piti tehdä, ja miten mikropalvelut kommunikoivat keskenään. Se auttoi myös päättämään, missä järjestyksessä mikropalvelut kehitetään.

5.4 Alustavat kehityksen toimenpiteet

Suunnittelun jälkeen alkoi alustavien toimenpiteiden tekeminen ennen varsinaisen kehityksen aloittamista. Hyödynsin kehityksessä GitHubin tarjoamia työkaluja ja palveluita. Alustaviin toimenpiteisiin kuului Git-repositorion ja Kanban-taulun luonnit. Nämä olivat projektin kehityksessä ja sen hallinnassa keskeisiä asioita ja niitä käytettiin koko kehityksen aikana aktiivisesti.

5.4.1 Git-repositorion luonti

Ensimmäiseksi loin projektille Git-repositorion versionhallintaa varten. Repositorion loin GitHub-palvelussa, johon voi tallentaa palvelimelle Git-repositorioita, jotta ne pysyvät siellä tallessa. GitHub on erittäin suosittu ja käytetty palvelu ammatillisessa ohjelmistokehityksessä. Se tarjoaa myös paljon muita ominaisuuksia ohjelmistokehitykseen kuin pelkän pilvitallennustilan Git-repositorioille.

Projektin lähdekoodin ja dokumenttien ylläpitämiseen käytin versionhallintaa. Ohjelmistokehityksessä versionhallinnalla tarkoitetaan ohjelmakoodin muutosten hallinnoimista ja seurantaa (Atlassian 2024). Sen avulla projektin lähdekoodiin tehdyt muutokset pysyvät hyvin ja järjestelmällisesti tallessa, sekä aikaisempia ohjelmiston versioita on mahdollista tarkastella ja käyttää. Projektin versionhallintajärjestelmänä käytin Gitiä. Git on nykyään suosituimpia versionhallintajärjestelmiä ohjelmistokehityksessä (Atlassian 2024). Valitsin Gitin sen suosion takia, ja koska minulla oli sen käyttämisestä jo paljon kokemusta.

Git-versionhallinnassa pusketaan muutoksia commiteilla repositorioihin. Repositorio on kuin iso säiliö, joka pitää sisällään koko versionhallinnan ja siihen tallennetut tiedostot. Commitit pitävät sisällään muutoksia esimerkiksi lähdekoodiin, jotka säilötään repositorioon. Jokainen commit pitää sisällään muutoksen ajankohdan, tekijän ja viestin sen tarkoituksesta tai sisällöstä (Atlassian 2024). Tämä auttaa pitämään versionhallintaa johdonmukaisena, jolloin tiedetään, milloin muutos on tehty ja kuka sen teki.

Git-versionhallinnassa voidaan myös tehdä haarakkeita, jotka pitävät sisällään eri version tallennetuista tiedostoista. Niitä käytetään yleensä, kun useat kehittäjät puskevat muutoksia samaan repositorioon, jolloin haarakkeita voidaan luoda eri ominaisuuksille ja kehittäjille helpottaen samanaikaista työstämistä (Atlassian 2024). Tässä projektissa käytin kuitenkin vain yhtä haaraketta repositoriota kohden, koska kehitin projektia yksin ja se yksinkertaisti kehitystä.

Versionhallintaa itsessään ei kuitenkaan voida varsinaisesti pitää kehittämismenetelmänä, koska sitä voidaan käyttää lukuisilla eri tavoilla osana ohjelmistokehitystä. Tässä projektissa yhtenä kehittämismenetelmänä toimi lähdekoodin ja tiedostojen usein puskeminen versionhallintajärjestelmään yhtä haaraketta käyttäen. Tarkoituksena oli puskea muutoksia pienissä erissä. Tällä pyrin saavuttamaan selkeän ja johdonmukaisen versionhallintahistoria, josta oli helppo tarkastella edistymistä.

GitHubista kloonasin paikallisen kopion Git-repositoriosta omalle tietokoneelleni. Näin pystyin tekemään muutoksia versionhallintaan omalla koneellani, jonka jälkeen pystyin puskemaan muutoksia GitHubin palvelimilla tallennettuun etäversioon repositoriostani. Jos oma koneeni olisi kehityksen aikana tuhoutunut, versionhallinta olisi pysynyt tallessa GitHubissa, josta olisin voinut kloonata uuden paikallisen kopion repositoriosta.

5.4.2 Kanban-taulun luonti

Git-repositorion luonnin ja alustuksen jälkeen loin projektille Kanban-taulun GitHub Projects -työkalulla. GitHub Projects on GitHub-palvelussa oleva ilmainen projektinhallintatyökalu, jolla voi esimerkiksi luoda Kanban-tauluja. Tauluun laitoin eri tehtävien vaiheita kuvaavia osioita. Tämän jälkeen kasasin tauluun alustavat tehtävät. Kanbanin ansiosta näitä tehtäviä pystyi myöhemmin tarpeen tullessa muuttamaan, lisäämään ja poistamaan. Pyrin tehtävien nimeämisessä selkeyteen ja järjestelmällisyyteen, jottei taulusta tullut sekava. Liitteessä 5 on projektin Kanban-taulu kehityksen alussa, johon on koottu tehtäviä kasaumaan, joista osa on valittu valmiiksi aloitettaviksi. Tämä taulu muuttui paljon projektin edetessä.

Käytin projektinhallinnassa ja työnkulun ohjauksessa Kanban-menetelmää. Valitsin Kanban-menetelmän projektiin, koska se sopi siihen hyvin. Kanban on luonteeltaan kevyt ja helppo käyttää. Lisäksi ketterän ohjelmistokehitysmenetelmän ansioista pystyin muuttamaan ja hallinnoimaan työnkulkua vielä suunnittelun jälkeen kehityksen aikana. Koska kehitin projektia yksin, en halunnut käyttää laajempia projektinhallintamenetelmiä, kuten Scrumia.

Kanban on nykyään yksi suosituimmista ohjelmistokehitysmenetelmistä ketterään ohjelmistokehitykseen. Sen tarkoituksena on järjestää työnkulku visuaalisesti Kanban-taululla, johon tulee työnkulun vaiheita kuvailevia tehtäviä. Kanbanin avulla voidaan dynaamisesti hallinnoida tehtäviä, jonka avulla kehitys nopeutuu ja siitä tulee tehokkaampaa. Tehtävät laitetaan taululla eri osioihin,

jotka kuvaavat tehtävien vaihetta. Kanbanissa keskitytään vain työhön, joka on käynnissä. (Radigan 2024.) Kanbanin avulla voidaan siis visualisoida projektin kulkua, josta näkee helposti, mikä on tekemättä ja mikä valmis. Kun jokin työnalla oleva tehtävä on valmis, se voidaan taululla siirtää valmiiden tehtävien osioon.

5.5 Protocol Buffersin kanssa työskentely projektissa

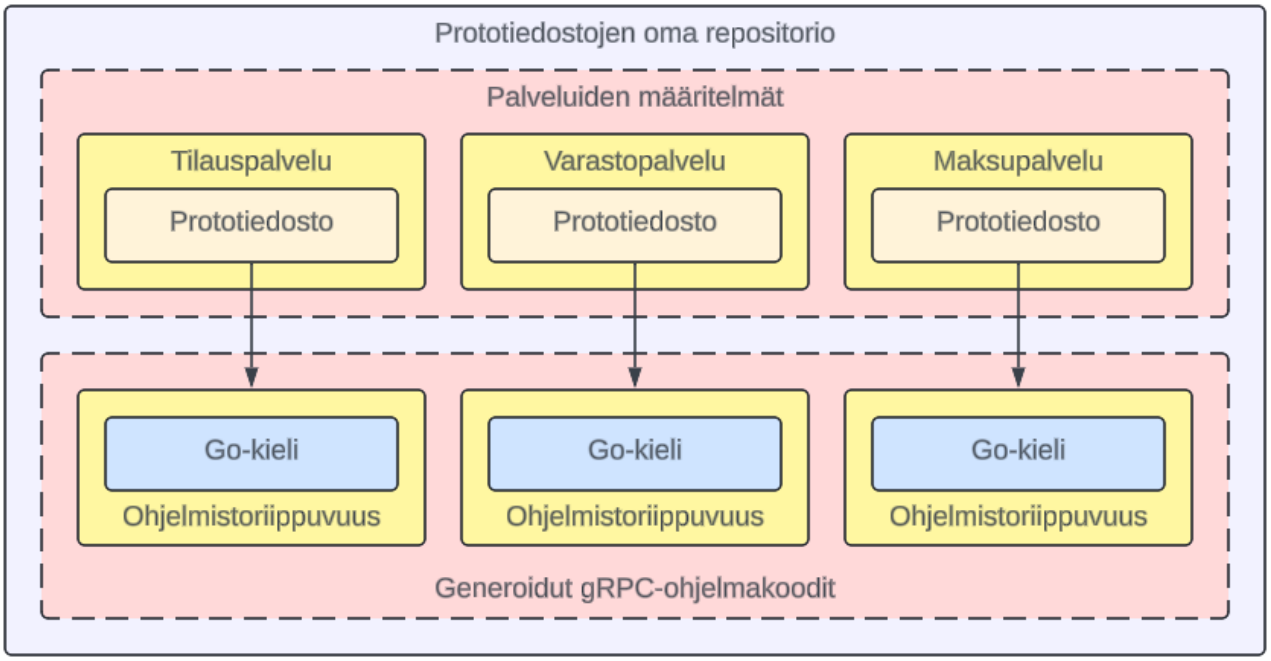
Ennen kuin lähdin ohjelmoimaan mikropalveluita, määrittelin ensin niiden gRPC API:t ja generoin tarvittavat gRPC-ohjelmakoodit gRPC API:en ohjelmoimiseksi. Tähän käytin Protocol Buffersia, koska sen käyttäminen gRPC API:en määritelmäkielenä ja datan sarjoittamismuotona on olennainen osa gRPC:n hyödyntämistä. gRPC API:en määritelmät järjestin prototiedostoihin, jotka tallensin versionhallintaan. Hyödynnettävyyden kannalta halusin projektissa järjestää prototiedostot hyvin ottamalla selvää parhaista käytänteistä.

5.5.1 gRPC API:en määritelmien kirjoittaminen ja prototiedostojen järjestäminen

Loin ensin projektin repositorioon hakemiston prototiedostoja varten, johon loin erillisen prototiedoston jokaista mikropalvelua varten. Jokaisessa prototiedostossa säilytin yhden palvelun määritelmät, jotta eri palveluiden määritelmät saatiin paremmin eristettyä toisistaan. Näin niitä oli helpompi hallita, kuin jos kaikkien palveluiden määritelmät olisi säilytetty yhdessä isossa tiedostossa. Prototiedostoihin kirjoitin määritelmät palveluiden etäproseduurikutsuista ja niissä käytettävistä viesteistä (Liite 6). Määrittelin kaikkien mikropalveluiden gRPC API:t kerralla, jotta oli helpompi hahmottaa, mitä viestejä ja dataa palveluiden välillä tullaan siirtämään etäproseduurikutsujen yhteydessä. Määrittelin vain yksinkertaiset etäproseduurikutsut, koska suoratoistoa ei projektissa käytetty. Koodin kirjoittamiseen käytin Neovim-nimistä tekstieditoria. Neovim on ilmainen ja avoimen lähdekoodin suosittu moderni komentorivipohjainen tekstieditori, joka laajentaa vanhempaa Vim-nimistä komentorivipohjaista tekstieditoria.

Prototiedostoja on kuitenkin mahdollista säilyttää myös omassa repositoriossa, jolloin palveluiden määritelmät ja generoidut gRPC-ohjelmakoodit ovat erillään itse palveluiden lähdekoodista. Tämä voi olla hyödyllistä, jos mikropalvelut kehitettäisiin eri ohjelmointikielillä eri repositorioissa, jolloin gRPC-ohjelmakoodit voitaisiin generoida kaikille tarvittaville ohjelmointikielille. Näin ne eivät olisi riippuvaisia mistään tietystä mikropalvelusta, vaan kaikkia palveluiden määritelmiä voitaisiin järjestelmällisesti säilyttää yhdessä niille tarkoitetussa paikassa. Esimerkiksi Go-ohjelmointikielellä toteutettu mikropalvelu tarvitsee ainoastaan gRPC-koodia Go-kielelle, eikä esimerkiksi Java-kielelle. Java-kieltä olisi turha ylläpitää tämän palvelun repositoriossa, koska kyseinen mikropalvelu ei käytä sitä. (Babal 2023, luku 3.3.)

Opittuani paremman tavan prototiedostojen järjestämiseen mikropalveluprojektissa, loin prototiedostoja varten oman Git-repositorion ja siirsin tiedostot sinne. Siellä niistä pystyi generoimaan gRPC-ohjelmakoodit Go-kielelle ohjelmistoriippuvuuksiksi ulkoisten repositorioiden käytettäväksi (Kuva 15). Tein repositorion alustamisen yhteydessä luvun 5.4.1 mukaisesti samat toimenpiteet, kuin projektin toisenkin repositorion kanssa.



Kuva 15. Projektin prototiedostojen ja gRPC-ohjelmakoodien säilyttäminen niille tarkoitetussa omassa repositoriossa

5.5.2 gRPC-ohjelmakoodien generointi

Kun sain mikropalveluiden gRPC API:t määriteltyä, oli aika generoida gRPC-ohjelmakoodit prototiedostoista. Ohjelmakoodit on mahdollista generoida Protocol Buffersin kääntäjällä komentorivillä yhdellä komennolla, mutta työskentelyn helpottamiseksi ja automatisoinnin saavuttamiseksi loin sitä varten skriptin, joka ajoi komennon tarvittavilla argumenteilla (Liite 7). Skriptin tallensin versionhallintaan, jotta se pysyi siellä tallessa ja kenen tahansa tarkasteltavissa. Kyseisen skriptin avulla ei tarvinnut käyttää suoraan Protocol Buffersin kääntäjäohjelmaa. Tämä helpotti työtä, koska skriptiin tallennettiin tarvittavat asetukset koodigeneroinnin tekemiseksi, jotta koodigenerointi toimi joka kerta samalla halutulla tavalla.

Skriptiä ajamalla pystyin generoimaan kaikista projektin prototiedostoista gRPC-ohjelmakoodit projektissa käytettävälle Go-ohjelmointikielelle prototiedostoissa määriteltyyn hakemistosijaintiin. Myöhemmin käytin näitä generoituja ohjelmakoodeja gRPC-asiakkaiden ja -palvelinten ohjelmointiin mikropalveluissa. Tein generoiduista gRPC-ohjelmakoodeista julkiset ohjelmistoriippuvuudet ja

versioin ne. Näin pystyin myöhemmin käyttämään niitä repositorion ulkopuolella mikropalveluissa. Go-kielellä tämä tapahtui suoraviivaisesti, koska sen avulla projekteista voi tehdä ohjelmistoriippuvuuksia suoraan GitHub-repositorioilla, joita voi helposti käyttää toisessa Go-projektissa.

Tein skriptin ajettavaksi Bash-komentotulkilla, koska käytin Debian-nimistä Linux-jakelua kehitysympäristönä Windowsilla WSL-nimisen teknologian kautta. WSL on lyhenne sanoista Windows Subsystem for Linux. Sillä on mahdollista ajaa eristettyjä Linux-käyttöjärjestelmäympäristöjä Windows-käyttöjärjestelmällä. Linux-ympäristössä pystyi tekemään monia kehitykseen liittyviä asioita helpommin ja nopeammin tehokkaan komentorivin avulla. Käyttämäni Neovim-koodieditori oli myös vaivattomampi käyttää Linuxilla.

5.6 Kehitettyjen mikropalveluiden toteutus

Käytin mikropalveluiden kehittämisen tukena 12-tekijän sovellusta. Se on menetelmä ohjelmistopalveluiden kehittämiseen millä tahansa ohjelmointikielellä, jossa pyritään seuraamaan kahtatoista eri tekijää, joiden mukaan sovellusta kehitetään (Wiggins 2017). Tällä menetelmällä pyrin tehostamaan omaa oppimistani ja mikropalveluiden hyödynnettävyyttä, koska menetelmä on suosittu ammattimaailmassa. Wiggins (2017) kuvailee kyseisen menetelmän tavoitteita seuraavasti:

- Sovellus järjestetään deklaratiivisesti, jolla helpotetaan muiden kehittäjien osallistumista projektiin.
- Sovellusta voidaan siirtää erinomaisesti eri suoritusympäristöjen välillä, jolloin sitä voidaan ajaa eri tietokoneilla muuttamatta lähdekoodia.
- Sovelluksen käyttöönotto soveltuu moderneille pilvipalvelualustoille, joissa voidaan käyttää esimerkiksi kontteja.
- Kehityksessä hyödynnetään jatkuvaa käyttöönottoa kehityksen nopeuttamiseksi.
- Sovellusta voidaan laajentaa ilman merkittäviä muutoksia työkaluihin, arkkitehtuuriin ja kehittämiskäytäntöihin.

5.6.1 Mikropalveluissa käytetty arkkitehtuurityyli

Ensimmäiseksi päätin käytettävän arkkitehtuurityylin jokaisen mikropalvelun rakentamiseen. En osannut ottaa tätä huomioon suunnitteluvaiheessa, koska keskityin siinä kehitettävien mikropalveluiden kommunikointiin, enkä niinkään yksittäisten mikropalveluiden yksityiskohtaiseen toteutukseen.

Otin selvää eri arkkitehtuurityyleistä, joita voidaan käyttää yksittäisten mikropalveluiden rakentamiseen, ja päädyin käyttämään kuusikulmaista arkkitehtuuria (englanniksi hexagonal architecture). Tässä projektissa kiinnitin erityistä huomiota arkkitehtuuriin, koska sain erinomaisen mahdollisuuden harjoitella oikeaoppisen ohjelmistoarkkitehtuurin käyttämistä. Ohjelmistoarkkitehtuuri tekee

lähdekoodin kirjoittamisesta ja hallinnoimisesta paljon järjestelmällisempää. Lisäksi se helpottaa ohjelmiston laajentamista kehityksen edetessä.

Kuusikulmainen arkkitehtuuri on vuonna 2005 Alistair Cockburnin esittämä ohjelmistojen arkkitehtuurityyli, jolla voidaan rakentaa toisistaan irti olevia sovelluksen komponentteja, jotka voidaan yhdistää toisiinsa porttien ja liittimien kautta. Käyttämällä portteja ja liittimiä voidaan eristää sovelluksen liiketoimintalogiikka ulkoisista riippuvuuksista. (Babal 2023, luku 4.1.) Tässä arkkitehtuurissa on ideana, että palveluihin tulevat pyynnöt tehdään sisään tuleviin liittimiin ja palveluista ulos lähtevät pyynnöt tehdään ulos lähteviin liittimiin. Liiketoimintalogiikka on näiden liittimien keskellä, jota ympäröi portit. Portit sisältävät metodeja, joiden ohjelmalogiikka toteutetaan liittimiin. (Richardson 2018, luku 2.1.2.) Kyseinen arkkitehtuuri on suosittu mikropalveluiden rakentamisessa. gRPC:n käyttäminen tekee kuusikulmaisen arkkitehtuurin toteuttamisesta helpompaa valmiina generoiduilla gRPC-ohjelmakoodeilla, joita voidaan käyttää liittimien toteuttamiseen (Babal 2023, luku 4.1.4).

5.6.2 Mikropalveluiden ohjelmointi

Aloitin mikropalveluiden ohjelmoinnin maksupalvelusta. Lähdin liikkeelle siitä, koska se on tilaustoiminnon viimeinen palvelu, johon lähetetään pyyntö. Oli siis järkevämpää kehittää mikropalvelut alhaalta ylös. Tilauspalvelun täytyy lähettää pyyntö kahteen muuhun palveluun, mutta maksupalvelun ei tarvitse lähettää pyyntöä mihinkään. Tämän takia tilauspalvelua ei olisi voinut ohjelmoida loppuun asti toimivaksi ennen varastopalvelua ja maksupalvelua kommunikoinnin synkronisen luonteen vuoksi.

Ensimmäiseksi alustin palvelun luomalla tarvittavat kansiot ja Go-moduulin. Go-moduulilla on helpompi kehittää Go-kielellä tehtyjä projekteja hallinnoimalla sen metadataa ja ohjelmistoriippuvuuksia go.mod-nimisessä tiedostossa. Tähän tiedostoon tallentui tiedot palvelussa käytettävistä ohjelmistoriippuvuuksista, kuten käyttämistäni generoiduista gRPC-ohjelmakoodeista (Liite 8). Asensin riippuvuudet sitä mukaan, kun niitä tarvitsin. Tämän jälkeen loin palvelun liiketoimintalogiikassa käytettävän maksuobjektin ja palvelun sisäisen API:n liiketoimintalogiikalle, jota ohjelmoitava gRPC API voi kutsua.

Seuraavaksi ohjelmoin sisään tulevan liittimen gRPC API:lle, joka sisälsi varsinaisen gRPC API:n koodin. Siinä käytin generoitua gRPC-palvelinkoodia pohjana, johon ohjelmoin maksupalvelun API:n sisältämän etäproseduurikutsun logiikan. Ohjelmoitavaan etäproseduurikutsun toteutukseen käytin generoidun gRPC-koodin sisältämiä pyyntö- ja vastausviestejä. Liittimelle ohjelmoin myös toiminnon, jolla gRPC-palvelin voitiin ajaa jossakin pistokkeessa pyyntöjen vastaanottamiseksi. (Liite 9.) Pistokkeen käyttämän portin tein konfiguroitavaksi ympäristömuuttujalla ja konfiguraatiotiedostolla, jotta palvelua voitaisiin ajaa helposti eri ympäristöissä. Ympäristömuuttujien

käyttäminen palvelun konfiguroimiseen on keskeinen osa 12-tekijän sovellusta. Mikropalvelun sai käynnistettyä komentoriviltä helposti yhdellä Go-kielen komentoriviohjelman komennon avulla.

Maksupalvelun jälkeen aloitin varastopalvelun kehityksen. Sen kanssa tein samat toimenpiteet, kuin maksupalvelun kanssa. Varastopalvelu ei lähettänyt pyyntöjä toisiin palveluihin, vaan tuli vastaanottamaan pyyntöjä tilauspalvelulta. Siihen ohjelmoin kolme eri etäproseduurikutsua. Varastopalvelun gRPC-palvelimen ja gRPC API:n ohjelmoin samalla tavalla kuin maksupalvelun. Tein varastopalveluun yksinkertaisen muistissa olevan tietokannan, johon tallensin testidataa tuotteista. Tämän avulla pystyin testaamaan tuotteita etäproseduurikutsuissa ja myöhemmässä vaiheessa olevaa valmista tilaustoimintoa, joka tuli käyttämään tätä testidataa tehtävissä pyynnöissä. Varastopalvelu tulosti tämän testidatan konsoliin, jotta oikeita tuotetunnuksia pystyi käyttämään pyyntöjen datassa.

Viimeiseksi kehitin tilauspalvelun. Tämä palvelu tuli lähettämään pyyntöjä varastopalveluun ja maksupalveluun, joten siihen minun piti ohjelmoida gRPC-asiakkaat molemmille palveluille pyyntöjen lähettämiseksi. Lisäksi ohjelmoin gRPC-palvelimen ja yhden etäproseduurikutsun sisältävän gRPC API:n tilauspyyntöjen tekemiseksi. gRPC-asiakkaaseen ohjelmoin koodin, jolla pystyi yhdistämään toiseen mikropalveluun. Lisäksi ohjelmoin tarvittavat metodit etäproseduurikutsujen lähettämiseen käyttäen generoituja gRPC-ohjelmakoodeja (Liite 10).

Ohjelmoin tarvittavien pyyntöjen ja vastausten käsittelyn tilauspalvelun liiketoimintalogiikan metodiin, jolla pystyi tekemään tilauksia (Liite 11). gRPC API:n sisältämä etäproseduurikutsu käytti tätä metodia. Ohjelmoin siihen myös aikarajan koko tilausoperaatiolle, jotta se ei odota ikuisesti, jos esimerkiksi maksupalvelu ei ikinä palauttaisi vastausta. Jos tilausoperaatio epäonnistui virheen takia, se palautti etäproseduurikutsusta virheviestin ja gRPC-tilakoodin. (Liite 12.)

Tilauspalvelun konfiguraatioihin lisäsin konfiguraatiot varastopalvelun ja maksupalvelun osoitteille ja porteille, jotta niihin pystyi yhdistämään. Osoitteet ja portit yhdistin pistokkeiksi, joita gRPC-asiakkaat käyttivät yhteyden luomiseen mikropalveluihin. En kovakoodannut näitä lähdekoodiin, koska se ei ole hyvä tapa 12-tekijän sovelluksessa.

5.7 Toteutettujen gRPC API:en testaus

gRPC API:en testaamiseen on olemassa useita eri työkaluja. Alkuperäinen suunnitelmani gRPC API:en testaamiseen oli käyttää Postman-nimistä työkalua. Postman on graafisen käyttöliittymän ohjelmisto, jolla on mahdollista testata eri API-tyyppejä, kuten gRPC:tä ja REST:iä. Projektin aikana kuitenkin opin, että gRPC API:en testaamiseen on olemassa suosittu komentoriviohjelma nimeltä grpcurl. Sen avulla voi tehdä gRPC-pyyntöjä gRPC-palveluihin kätevästi ajamalla komentoja suoraan komentoriviltä. Käytin kyseistä työkalua gRPC API:en testauksessa.

Jotta pystyin työkalulla testata pyyntöjen lähettämistä mikropalveluihin helposti, mikropalveluiden täytyi ottaa päälle gRPC-palvelimen heijastusominaisuus. gRPC:n heijastus on protokolla, jota gRPC-palvelimet voivat käyttää Protocol Buffers -määritelmien julkaisemiseen standardoidun etäproseduurikutsupalvelun yli (gRPC Authors 2024b). Toisin sanoen heijastus mahdollistaa etäproseduurikutsujen tekemisen ilman prototiedostojen läsnäoloa gRPC-asiakkaan puolella, mikä helpottaa pyyntöjen testausta.

Ohjelmoin heijastuksen mikropalveluihin, jolloin se oli päällä vain kehitystilassa, koska se on tarkoitettu testaukseen ja debuggaukseen. Jos gRPC API on saatavilla julkisille käyttäjille, gRPC:n heijastusta ei kannata käyttää, koska sitä voidaan pitää tietoturvaongelmana (gRPC Authors 2024b). Tämän jälkeen pystyin grpcurl-ohjelmalla lähettämään gRPC-pyyntöjä gRPC-palvelimille ilman prototiedostoja. grpcurlilla oli mahdollista määritellä pyyntöviestien data helpommin käsiteltävässä JSON-muodossa, jonka se sarjoitti Protocol Buffers -muotoon ennen pyynnön lähettämistä. Vastauksena se tulosti vastausviestin datan tai virheen sattuessa virheen.

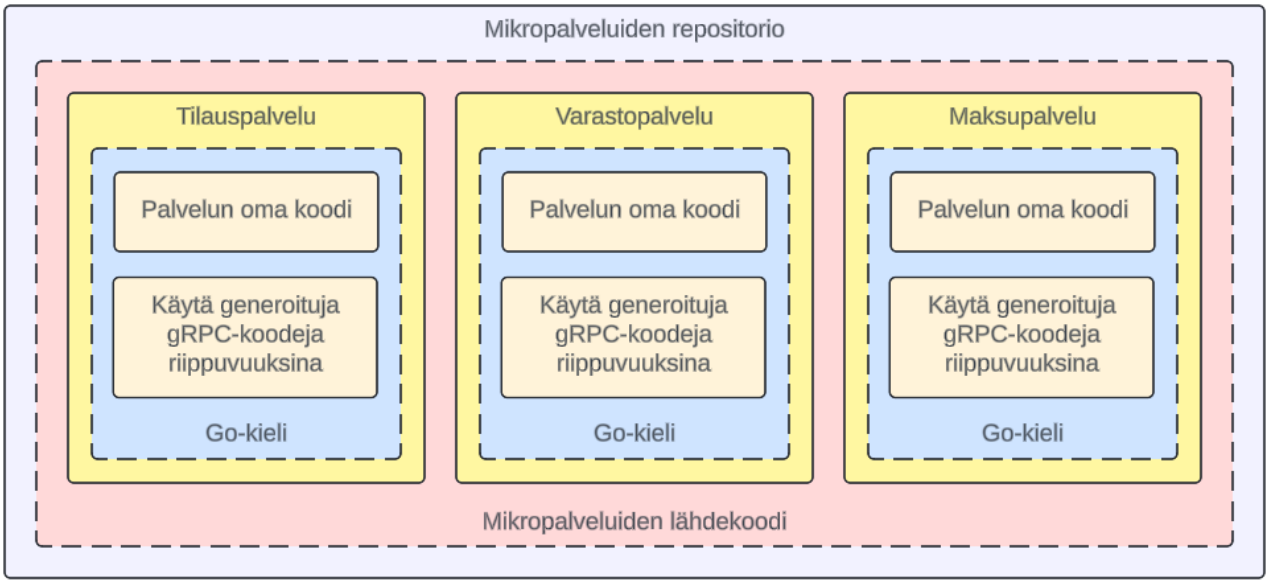
Tein mikropalveluiden repositorioon valmiita JSON-tiedostoja tuotetilaustoiminnon pyyntöjen testaamisen helpottamiseksi. Näiden tiedostojen testidatat käyttivät varastopalvelun luomien testituotteiden tuotetunnuksia. Näin esimerkiksi muut projektia käyttävät voivat testata pyyntöjä nopeammin ja vaivattomammin. Liitteessä 13 on onnistuneen etäproseduurikutsun testaus tilauspalveluun. Liitteessä 14 on epäonnistuneen etäproseduurikutsun testaus tilauspalveluun, joka palauttaa virheen sattuessa virhettä kuvaavan gRPC-tilakoodin ja virheviestin. Mikropalvelut tulostivat lokeja tehtävistä pyynnöistä, joita seuraamalla tiesin, että gRPC-kommunikointi toimi. Niiden avulla myös tiesin, jos pyynnöt tulivat tilauspalvelulta perille maksupalveluun ja varastopalveluun.

5.8 Projektin dokumentointi ja tulokset

Projektin lopuksi vielä dokumentoin sen. Dokumentoin projektin englanniksi sen Git-repositorioihin Markdown-tiedostomuodossa kohderyhmää ajatellen, koska projektista voi hyötyä ohjelmistokehittäjät ympäri maailmaa. Tästä syystä en nähnyt järkevänä kirjoittaa dokumentaatiota suomeksi. Dokumentaatioon kirjoitin esimerkiksi ohjeita siitä, miten projektia voidaan käyttää. Se sisälsi mikropalveluiden ajamisen, niiden konfiguroimisen ja gRPC API:en testauksen. Projekti dokumentoitiin, jotta projektin ulkopuoliset henkilöt myös ymmärtävät projektista ja voivat hyötyä siitä enemmän.

Lopputuloksena projektissa syntyi Git-repositorio kolmen mikropalvelun lähdekoodeille sekä erillinen Git-repositorio prototiedostoille ja generoiduille gRPC-ohjelmakoodeille. Nämä julkaistiin avoimena lähdekoodina GitHubissa. Mikropalvelut ohjelmoitiin kaikki Go-ohjelmointikielellä kuusikulmaista ohjelmistoarkkitehtuuria seuraten. Ne käyttivät prototiedostoista generoituja gRPC-

ohjelmakoodeja ohjelmistoriippuvuuksina gRPC-asiakkaiden, gRPC-palvelinten ja gRPC API:en ohjelmoimiseksi (Kuva 16).



Kuva 16. Mikropalveluiden lähdekoodit niiden Git-repositoriossa

Mikropalveluiden tueksi ei tehty erillisiä komponentteja, vaan ne jätettiin yksittäisiksi palveluiksi, joiden ajamiseksi ei tarvinnut esimerkiksi erillistä tietokantapalvelinta. Varastopalvelu sisälsi tietokoneen muistissa olevan tietokannan, johon mikropalvelu tallensi testidataa tuotteista palvelun käynnistyessä. Mikropalveluiden Git-repositorioon tallennettiin myös valmiita testidataa sisältäviä JSON-tiedostoja gRPC-pyyntöjen testaamiseksi. Lisäksi Git-repositorioihin kirjoitettiin Markdown-muodossa dokumentaatiot olennaisista ja tärkeistä projektiin liittyvistä asioista.

6 Pohdinta

Tässä luvussa pohdin opinnäytetyöprosessia ja sen osana toteutettua ohjelmistoprojektia. Käyn läpi työn onnistumista, projektissa käytettyjen kehittämismenetelmien toimivuutta, jatkokehitysmahdollisuuksia sekä omaa oppimistani koko työn aikana. Lisäksi pohdin omia onnistumisia ja epäonnistumisia, sekä työn hyödynnettävyyttä kohderyhmä huomioon ottaen.

6.1 Työn onnistuminen

Luvussa 1 määritettiin tämän opinnäytetyön tavoitteet. Työn päätavoitteena oli ottaa selvää gRPC:n käyttämisestä ja hyödyntämisestä mikropalveluiden synkronisessa kommunikoinnissa. Tähän tavoitteeseen päästiin hyvin. Teorian tueksi toteutettu käytännön ohjelmistoprojekti tuki tämän tavoitteen saavuttamista enemmän, kuin jos se olisi jätetty pois. Teoriasta ei olisi kauheasti hyötyä, jos gRPC-kommunikointia ei osattaisi toteuttaa käytännössä, koska sen toteuttaminen ohjelmistoihin on sen käytön tarkoitus. Käytännön näkökulma siis paransi työn hyödynnettävyyttä ja uskottavuutta. Synkronisessa kommunikoinnissa onnistuttiin pysymään työn rajauksen kannalta suurimmaksi osaksi hyvin, eikä gRPC:n osalta keskitytty liikaa sen yksityiskohtiin niin kuin oli tarkoituskin.

Tietoperustaan onnistuin laatimaan kattavan teorian aiheeseen liittyen hyödyntämällä omaa tietämystäni ja kokemustani, sekä käyttämällä ajankohtaista materiaalia, joista suurimman osan on kirjoittanut alan ammattilaiset. Käyttämäni kirjat ovat kirjoittaneet ammattilaiset, jotka ovat työskennelleet mikropalveluiden ja gRPC:n kanssa ammatissaan. Heidän näkemyksensä siis lisäsivät työn luotettavuutta ja toivat siihen paljon hyödyllistä tietoa.

Työn toinen tavoite oli oppia lisää aiheesta ja käsiteltävistä teemoista, sekä kehittää ammatillisia ohjelmistokehitystaitoja ja -tietoja. Tähän tavoitteeseen päästiin erittäin onnistuneesti. Mikropalvelut ovat luonteeltaan hyvin laaja ja haastava aihealue. Minun piti opiskella aihetta paljon ennen kuin pystyin edes aloittamaan varsinaisen kirjoittamisen. Lukuisten tuntien käyttäminen opiskeluun koko opinnäytetyön aikana oli kuitenkin antoisa, ja opin aiheesta sekä siihen liittyvistä asioista paljon uutta, josta uskon olevan hyötyä muuallakin kuin mikropalveluiden kehityksessä. Toteutettu ohjelmistoprojekti kasvatti taitojani ja tietämystäni mikropalveluiden kehittämisestä sekä gRPC:n käyttämisestä. Lisäksi opin esimerkiksi kuusikulmaisen arkkitehtuurin käytöstä ja 12-tekijän sovelluksesta.

Projektissa onnistuttiin toteuttamaan kaikki kolme mikropalvelua. Niille myös onnistuttiin toteuttamaan gRPC API:t ja mikropalveluihin jakautuva tilaustoiminto saatiin toteutettua alkuperäisen suunnitelman mukaan ongelmitta. Osa suunnitelmista kuitenkin muuttui projektin edetessä ja siitä karsittiin kohtia pois, jotka eivät olleet projektin onnistumisen kannalta olennaisia. API-

yhdyskäytävä ja ulkoiset tietokannat jätettiin pois. En nähnyt järkevänä toteuttaa niitä, koska niitä ei projektissa tarvittu ja ne olisivat voineet mennä työn rajauksen ulkopuolelle.

Projektissa ohjelmoin yksinkertaisen virheenkäsittelyn mikropalveluille ja tilaustoiminto palautti vain yhden tilakoodin. Tuotannollinen mikropalveluiden virheenkäsittely on monimutkaisempi ja olisi vaatinut paljon enemmän työtä sekä koodia. gRPC:n eri tilakoodien avulla voidaan kuitenkin virheenkäsittelystä saada hyvin luotettava, koska jokainen tilakoodi kuvaa eri virhetyyppiä. Toiset mikropalvelut voivat käyttää näitä tilakoodeja päättääkseen, mitä eri virhetyyppien sattuessa tehdään.

gRPC:n ymmärtäminen vaatii paljon tietoteknistä ymmärtämistä, jota ei kuitenkaan pelkällä lähdekoodin lukemisella voi korvata. Tästä syystä gRPC:n ymmärtäminen vaatii myös paljon teoriaa. Toteutettu projekti kuitenkin antaa hyvän kuvan, miten gRPC:tä voidaan käyttää käytännössä mikropalveluiden synkronisessa kommunikoinnissa. Kun sen yhdistää työn teorian kanssa, saadaan aikaiseksi hyvä kokonaisuus ilmiöstä, jota ohjelmistokehittäjät voivat hyödyntää pohtiessaan gRPC:n käyttämistä omissa sovelluksissa.

On kuitenkin hyvä huomata, että projekti ohjelmoitiin Go-kielellä. Kaikkia sovelluksia ei kirjoiteta tällä ohjelmointikielellä, joten teknilliset yksityiskohdat eroavat eri ohjelmointikielien ja niiden ekosysteemien välillä. Siitä kuitenkin nähdään yleisellä tasolla gRPC:n pääkomponenttien toteuttaminen ja gRPC API:en testaus. Kirjoittamani lähdekoodi projektissa ei ole paras mahdollinen, koska en ole Go-kielen ammattilainen, mutta sitä on mahdollista jatkossa muokata ja parantaa.

Projektissa käytettiin HTTP/2-protokollaa gRPC-kommunikoinnissa, mutta gRPC:tä on mahdollista käyttää myös HTTP/3-protokollalla. HTTP/3 toimii teknillisestä näkökulmasta eri tavalla kuin HTTP/2, mutta sillä on mahdollista saada aikaiseksi vielä nopeampi kommunikointi. gRPC ei kuitenkaan tällä hetkellä tue sitä kovin laajasti, koska gRPC rakennettiin alun perin käyttämään HTTP/2-protokollaa. Tulevaisuudessa HTTP/3:n käyttäminen gRPC-protokollassa saattaa saada paremman tuen eri ohjelmointikielille, mutta tähän voi mennä aikaa. Käyttämälläni Go-kielellä ei esimerkiksi ollut virallista ohjelmistokirjastoa sille.

6.2 Kehittämismenetelmien toimivuus

Projektissa käytetyt kehittämismenetelmät toimivat hyvin. Git-versionhallinnassa puskin lisäyksiä mikropalveluiden lähdekoodiin pienissä erissä niin kuin oli tarkoituskin. Menetelmällä vältettiin se, ettei esimerkiksi yhden mikropalvelun koko lähdekoodia puskettu yhdessä erässä. Tämä olisi tehnyt muutoksien tarkastelusta haastavaa ja epäjohdonmukaista.

Kanban toimi erinomaisena projektinhallintamenetelmänä. Alkuperäiset suunnitelmani muuttuivat jonkin verran projektin edetessä, joten Kanbanilla oli helppo muuttaa Kanban-tauluun laitettuja tehtäviä milloin tahansa. Se myös teki kehityksestä joustavampaa, koska projektia ei tarvinnut työstää sprinteissä. Sitä saatiin kehittää rauhassa oman aikataulun mukaan. Minulla oli tapana kerätä tietty määrä tehtäviä aina yhdelle mikropalvelulle kerralla. Aina kun sain yhden mikropalvelun valmiiksi, tein pienen yleiskatsauksen projektiin ja tarkastelin mitä olin saanut aikaiseksi. Tämä teki kehityksestä sujuvaa ja johdonmukaista.

12-tekijän sovellus ei toiminut suurena kehittämismenetelmänä projektissa, vaan oli enemmänkin tukena mikropalveluiden ohjelmoinnissa. Sen mukaan mikropalveluista tehtiin ajettavia eri ympäristöjen välillä ympäristömuuttujien ja konfiguraatiotiedostojen avulla. Menetelmä pitää sisällään useita tekijöitä liittyen käyttöönottoon, mutta näihin ei keskitytty, koska mikropalveluita ei otettu oikeaan käyttöön erillisille palvelimille. Ohjelmoin mikropalvelut kuitenkin niin, että tämä olisi mahdollista tulevaisuudessa muuttamatta lähdekoodin rakennetta merkittävästi. Tulevaisuudessa ne voidaan kontittaa helposti ja ottaa käyttöön esimerkiksi Kubernetes-alustalle harjoittelua ja oppimista varten.

Kuusikulmainen arkkitehtuuri ei toiminut projektissa kehittämismenetelmänä, vaan se toimi tapana, jonka mukaan mikropalveluiden lähdekoodi kirjoitettiin ja järjestettiin. Kyseinen arkkitehtuuri oli hyvä valinta kehitetyille mikropalveluille ja teki niiden lähdekoodeista yhtenäisiä sekä loogisia kokonaisuuksia. Niitä on huomattavasti helpompi laajentaa ja muuttaa, jos niitä halutaan jatkokehittää. Projektin jälkeen ymmärsin arkkitehtuurityylin käyttämisen tarkoituksesta ja hyödyistä enemmän ohjelmistoa rakentaessa, joten sen käyttäminen oli hyvä päätös.

6.3 Jatkokehittäminen

Jatkokehittämismahdollisuuksia projektille on hyvin paljon. Projekti kehitettiin minimaaliseksi tämän opinnäytetyön rajauksen vuoksi, joten se ei sisällä esimerkiksi mikropalveluiden kontittamista, ulkoisia tietokantoja, yhteyksien salausta eikä automaattisia testejä. Nämä ovat erinomaisia jatkokehittämismahdollisuuksia projektille, joista on mahdollista myös oppia lisää. Konttien kannalta voitaisiin esimerkiksi harjoitella Kuberneteksen käyttöä, koska se on nykyään käytetyimpiä alustoja mikropalveluiden käyttöönotossa konttien avulla. Koska tein mikropalveluista konfiguroitavia, tämä ei vaadi niiden lähdekoodiin merkittäviä muutoksia. Mikropalveluihin on myös mahdollista lisätä uusia ominaisuuksia ja lisäkonfiguraatioita helposti tarpeen mukaan oikeaoppisen arkkitehtuurin ansiosta. Kuusikulmainen arkkitehtuuri antaa mahdollisuuden kirjoittaa laadukkaita testejä mikropalveluille.

Synkronisen kommunikoinnin ja gRPC:n osalta projektia voidaan myös laajentaa paljon. Projektissa ei käytetty kaikkia gRPC:n ominaisuuksia, joista tuotannollisissa mikropalveluissa hyötyisi. Etäproseduurikutsujen mukana kulkevaa metadataa ei esimerkiksi käytetty, koska sille ei ollut tarvetta. Kommunikointi palveluiden välillä voitaisiin myös salata TLS-protokollan avulla. Käyttämälläni Go-kielen gRPC-kirjastolla oli tähän synnynnäinen tuki, jolla sen olisi saanut toteutettua melko suoraviivaisesti.

Lokeja projektissa ei myöskään käytetty hyvin. gRPC:n avulla on mahdollista keskittää lokien kirjoittaminen yhteen paikkaan lähdekoodissa, jolloin yhdessä funktiossa voidaan esimerkiksi kirjoittaa debug-lokeja kaikista etäproseduurikutsuista. Näin jokaiseen etäproseduurikutsuun ei tarvitse erikseen ohjelmoida lokitusta. Tästä on hyötyä, jos lokien kirjoittamisen logiikkaan liittyy useita tarkistuksia, kuten lokitason tarkistus. Ei olisi järkeä kopioida samaa logiikkaa kaikkiin etäproseduurikutsuihin, jos niitä olisi esimerkiksi kymmeniä.

Tilaustoimintoon voitaisiin myös ohjelmoida parempi virheenkäsittely, jossa hyödynnetään eri gRPC-tilakoodeja. Lisäksi siihen on mahdollista ohjelmoida toiminto, jolla epäonnistuneita pyyntöjä muihin palveluihin yritettäisiin uudestaan. Projektissa tein vain yksinkertaisen esimerkin, joka palauttaa virheen, jos pyyntö esimerkiksi maksupalveluun epäonnistuu hitaan yhteyden takia. Tuotannollisessa kommunikoinnissa tämä on hyvin köyhä ratkaisu. Projektia ei tehty tuotannolliseksi, joten sillä ei ole sisimmissään väliä. Hyödynnettävyyden kannalta se voisi kuitenkin olla hyvä, jotta esimerkkiprojekti kuvaisi paremmin oikean maailman tarpeita. Tulen todennäköisesti vielä jatkamaan projektin kehitystä, koska se toimii minulle hyvänä oppimisprojektina, jossa gRPC:tä, synkronista kommunikointia ja mikropalveluita voi harjoitella käytännössä.

6.4 Oma oppiminen

Oppimisen kannalta tämä opinnäytetyö oli erittäin antoisa. Opin paljon uutta ja samalla kasvatin jo olemassa olevaa tietämystäni työn teemoihin liittyen. Aihe oli luonteeltaan melko haastava, mutta olin siitä jo tietoinen ennen työn aloittamista. Tästä syystä se vaati paljon itsenäistä opiskelua ja oma-aloitteista asioista selvää ottamista. Olin kuitenkin aiheesta kiinnostunut, joten välillä työskentely oli jopa mukavaa. Omien uratavoitteideni kannalta sain työstä monia hyödyllisiä asioita irti, joista uskon olevan apua tulevaisuudessani ohjelmistoalalla. Toki kaikkia käsiteltäviä asioita en opiskellut syvällisesti, mutta niistä on hyvä jatkaa. Työ antoi minulle uusia näkemyksiä ohjelmistokehityksestä ja palvelinohjelmistojen kommunikoinnin tärkeydestä.

Opinnäytetyön aikana suunnitelmat muuttuivat välillä ja jouduin muuttamaan raportin rakennetta useita kertoja. Se ei mennyt alkuperäisen suunnitelman mukaan. Minulla oli alussa haasteita saada aikaan johdonmukainen ja hyvä rakenne työlle. Ajan myötä, kun opin aiheesta lisää,

rakenne alkoi kuitenkin muotoutua paremmaksi sisällöt mukaan lukien. Jatkuva oppiminen työtä tehdessä oli siis minulle erittäin tärkeä osa työtä. Aikataulussa kuitenkin onnistuin pysymään erinomaisesti ja sen kanssa ei tullut missään vaiheessa ongelmia.

Minun olisi työn alussa kannattanut kiinnittää enemmän huomiota rakenteeseen, jotta sitä ei olisi tarvinnut useita kertoja muuttaa. Tämä olisi kuitenkin vaatinut laajan tietämyksen aiheesta jo työn aloitushetkellä. Tuotoksen suunnittelun olisin myös voinut tehdä paremmin, koska sen suunnitelma muuttui kehityksen aloittamisen jälkeen. Yritin tehdä projektista aluksi liian laajan ja siihen olisi tullut paljon projektin tavoitteeseen nähden olemattomia asioita.

gRPC:n ekosysteemi on jatkuvassa kehityksessä ja muutaman vuoden kuluttua se saattaa olla vielä parempi kuin nyt. Lisäksi Protocol Buffers, joka on gRPC:n hyödyntämisen olennainen osa, on viime vuosina kehittynyt paljon. Tämän työn loppupuolella opin, että sille on tällä hetkellä olemassa suosittu ja aktiivisessa kehityksessä oleva työkalu nimeltä Buf, jonka avulla on mahdollista työskennellä prototiedostojen kanssa huomattavasti paremmin. Tämän työkalun suosio on noussut nopeasti ja se on saanut positiivisen vastaanoton.

6.5 Loppusanat

Tämä opinnäytetyö keskittyi mikropalveluiden synkroniseen kommunikointiin, joten se antoi vain pintaraapaisun mikropalveluarkkitehtuuriin ja kaikkiin mahdollisiin kommunikointitapoihin. gRPC:n kaikkia ominaisuuksia ja yksityiskohtia ei myöskään käsitelty, koska se on hyvin laaja teknologia. Siitä riittäisi käsiteltävää jopa useammalle eri työlle. Lisäksi synkroninen kommunikointi pitää sisällään paljon edistyneitä tekniikoita, joita tuotannollisissa sovelluksissa tulisi käyttää. Esimerkkeinä voidaan pitää oikeaoppista virheenkäsittelyä, salausta ja verkkovian takia epäonnistuneiden pyyntöjen uudelleenyrittämistä. Tässä työssä ei ollut tarkoitus syventyä niihin.

Aiheen kannalta opinnäytetyöni oli melko uniikki, koska en löytänyt internetistä kovin montaa samankaltaista opinnäytetyötä ainakaan suomeksi. Lisäksi gRPC:stä on tällä hetkellä melko vähän laadukasta tietoa ja tutkimuksia suomeksi. Tämän kannalta työ oli ainakin hyödyllinen ja tuottaa lisää hyödyllistä tietoa aiheesta. Englanniksi aiheesta löytyy paremmin tietoa kansainvälisen luonteensa takia ja se olikin yksi syistä, miksi käytin suurimmaksi osaksi kansainvälistä materiaalia lähteissä.

Aihetta tulen todennäköisesti jatkossa opiskelemaan vielä lisää laajemmin. Tuotos antaa hyvän ympäristön mikropalveluiden ja gRPC:n käytännön harjoittelemiseen, sekä se toimii hyvänä projektina oppimisen tukena. Kohderyhmää työ palvelee melko hyvin ja gRPC:stä tietämättömät oppivat siitä paljon uutta. Työ avasi aihetta hyvin ja siitä saadaan kattava kokonaiskuva gRPC:n hyödyntämisestä sekä käytöstä mikropalveluissa.

Lähteet

Acharya, D. 22.6.2022. Microservices vs APIs: Understanding the Difference. Kinsta Blog. Blogi. Luettavissa: https://kinsta.com/blog/microservices-vs-api/. Luettu: 6.6.2024.

Amazon Web Services 2024a. What is the Difference Between SOAP and REST? Luettavissa: https://aws.amazon.com/compare/the-difference-between-soap-rest/. Luettu: 6.6.2024.

Amazon Web Services 2024b. What's the Difference Between RPC and REST? Luettavissa: https://aws.amazon.com/compare/the-difference-between-rpc-and-rest/. Luettu: 6.6.2024.

Amazon Web Services 2024c. What's the Difference Between gRPC and REST? Luettavissa: https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/. Luettu: 10.6.2024.

Atlassian 2024. What is version control? Luettavissa: https://www.atlassian.com/git/tutorials/what-is-version-control. Luettu: 11.6.2024.

Babal, H. 2023. gRPC Microservices in Go. Manning Publications. Shelter Island, New York. E-kirja. Luettu: 8.5.2024.

Borysov, A. & Gardiner, R. 3.9.2021. Practical API Design at Netflix, Part 1: Using Protobuf FieldMask. Netflix TechBlog. Blogi. Luettavissa: https://netflixtechblog.com/practical-api-design-at-netflix-part-1-using-protobuf-fieldmask-35cfdc606518. Luettu: 24.5.2024.

Broshar, A. 23.3.2021. Service Discovery: Solving the Communication Challenge in Microservice Architecture. Koyeb Blog. Blogi. Luettavissa: https://www.koyeb.com/blog/service-discovery-solving-the-communication-challenge-in-microservice-architectures. Luettu: 22.5.2024.

Buchanan, I. 2024. Containers vs virtual machines. Luettavissa: https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms. Luettu: 27.6.2024.

Cloudflare 2024. HTTP/2 vs. HTTP/1.1: How do they affect web performance? Luettavissa: https://www.cloudflare.com/en-gb/learning/performance/http2-vs-http1.1/. Luettu: 14.6.2024.

gRPC Authors 2024a. About gRPC. Luettavissa: https://grpc.io/about/. Luettu: 14.5.2024.

gRPC Authors 2022. Core concepts, architecture and lifecycle. Luettavissa: https://grpc.io/docs/what-is-grpc/core-concepts/. Luettu: 4.6.2024.

gRPC Authors 2024b. Reflection. Luettavissa: https://grpc.io/docs/guides/reflection/. Luettu: 26.7.2024.

Google s.a. Build simple, secure, scalable systems with Go. Luettavissa: https://go.dev/. Luettu: 31.7.2024.

Google 2024. Overview. Luettavissa: https://protobuf.dev/overview/. Luettu: 24.5.2024.

Hillpot, J. 20.5.2024. REST APIs vs Microservices: Key differences. DreamFactory Blog. Blogi. Luettavissa: https://blog.dreamfactory.com/restful-api-and-microservices-the-differences-and-how-they-work-together/. Luettu: 19.6.2024.

Hillpot, J. 12.7.2023. What Are Containerized Microservices? DreamFactory Blog. Blogi. Luettavissa: https://blog.dreamfactory.com/what-are-containerized-microservices/. Luettu: 7.6.2024.

Indrasiri, K. & Kuruppu, D. 2020. gRPC: Up and Running. O'Reilly Media. Sebastopol, California. E-kirja. Luettu: 8.5.2024.

Kong 27.6.2022. Ultimate Guide: What are Microservices? Kong Blog. Blogi. Luettavissa: https://konghq.com/blog/learning-center/what-are-microservices. Luettu: 9.5.2024.

Kong 27.4.2022. What is gRPC? Kong Blog. Blogi. Luettavissa: https://konghq.com/blog/learning-center/what-is-grpc. Luettu: 14.5.2024.

Kong 10.3.2022. What is Service Discovery in Microservices? Kong Blog. Blogi. Luettavissa: https://konghq.com/blog/learning-center/service-discovery-in-a-microservices-architecture. Luettu: 23.5.2024.

Kuruppu, D. 31.8.2021. gRPC: A Deep Dive into the Communication Pattern. The New Stack. Luettavissa: https://thenewstack.io/grpc-a-deep-dive-into-the-communication-pattern/. Luettu: 10.5.2024.

Microsoft 2022. Communication in a microservice architecture. Luettavissa: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture. Luettu: 16.5.2024.

Mohan, N. 19.7.2021. Think gRPC, when you are architecting modern microservices! Cloud Native Computing Foundation Blog. Blogi. Luettavissa: https://www.cncf.io/blog/2021/07/19/think-grpc-when-you-are-architecting-modern-microservices/. Luettu: 10.6.2024.

Monteclaro, A. 7.11.2023. What Is a Network Server and How Does It Work? | ServerWatch. ServerWatch. Luettavissa: https://www.serverwatch.com/servers/network-server/. Luettu: 18.6.2024.

Newman, S. 2015. Building Microservices. 5. Painos. O'Reilly Media. Sebastopol, California. E-kirja. Luettu: 5.6.2024.

Newman, S. 2021. Building Microservices, 2nd Edition. 2. painos. O'Reilly Media. Sebastopol, California. E-kirja. Luettu: 9.5.2024.

Nord Security 2024. Binary Format. Luettavissa: https://nordvpn.com/fi/cybersecurity/glossary/binary-format/. Luettu: 27.5.2024.

Oracle 2022. What Is a Socket? Luettavissa: https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html. Luettu: 18.6.2024.

Ozkaya, M. 7.9.2021. Microservices Communications. Medium. Luettavissa: https://medium.com/design-microservices-architecture-with-patterns/microservices-communications-f319f8d76b71. Luettu: 20.5.2024.

The Postman Team 20.11.2023. gRPC vs. REST. Postman Blog. Blogi. Luettavissa: https://blog.postman.com/grpc-vs-rest/. Luettu: 27.5.2024.

The Postman Team 13.11.2023. What is gRPC? Postman Blog. Blogi. Luettavissa: https://blog.postman.com/what-is-grpc/. Luettu: 13.6.2024.

PowerCert Animated Videos 5.2.2020. What is a Server? Servers vs Desktops Explained. Video. Katsottavissa: https://www.youtube.com/watch?v=UjCDWCeHCzY. Katsottu: 18.6.2024.

Radigan, D. 2024. Kanban. Luettavissa: https://www.atlassian.com/agile/kanban/boards. Luettu: 5.7.2024.

Red Hat 2023. What are microservices? Luettavissa: https://www.redhat.com/en/topics/microservices/what-are-microservices. Luettu: 18.6.2024.

Richardson, C. 2018. Microservices Patterns. Manning Publications. Shelter Island, New York. E-kirja. Luettu: 8.5.2024.

Sandoval, K. 21.2.2024. What is Type Safety? Genezio Blog. Blogi. Luettavissa: https://genezio.com/blog/what-is-type-safety/. Luettu: 12.8.2024.

Shaw, K. 3.5.2024. What is a virtual machine, and why are they so useful? Network World. Luettavissa: https://www.networkworld.com/article/969185/what-is-a-virtual-machine-and-why-are-they-so-useful.html. Luettu: 18.6.2024.

Shuiskov, A. 2022. Microservices with Go. Packt Publishing. Birmingham. E-kirja. Luettu: 8.5.2024.

Sundar, A. 21.7.2023. Microservice Communication: A Complete Guide 2024. SayOne Blog. Blogi. Luettavissa: https://www.sayonetech.com/blog/microservices-communication/. Luettu: 5.6.2024.

Tanpure, P. 7.7.2023. Inter-Service Communication in Serverless Micro-Service Architecture. Medium. Luettavissa: https://medium.com/@pranav280300/inter-service-communication-in-micro-service-architecture-f91ffeae339d. Luettu: 20.5.2024.

Wiggins, A. 2017. The Twelve-Factor App. Luettavissa: https://12factor.net/. Luettu: 16.7.2024.

Xu, A & Lam, S. 15.12.2022. FANG Interview Question | Process vs Thread. ByteByteGo. Video. Katsottavissa: https://www.youtube.com/watch?v=4rLW7zg21gI. Katsottu: 24.7.2024.

Xu, A. & Lam, S. 11.10.2022. What Are Microservices Really All About? (And When Not To Use It). ByteByteGo. Video. Katsottavissa: https://www.youtube.com/watch?v=lTAcCNbJ7KE. Katsottu: 4.7.2024.

Xu, A & Lam, S. 1.12.2022. What is RPC? gRPC Introduction. ByteByteGo. Video. Katsottavissa: https://www.youtube.com/watch?v=gnchfOojMk4. Katsottu: 4.7.2024.

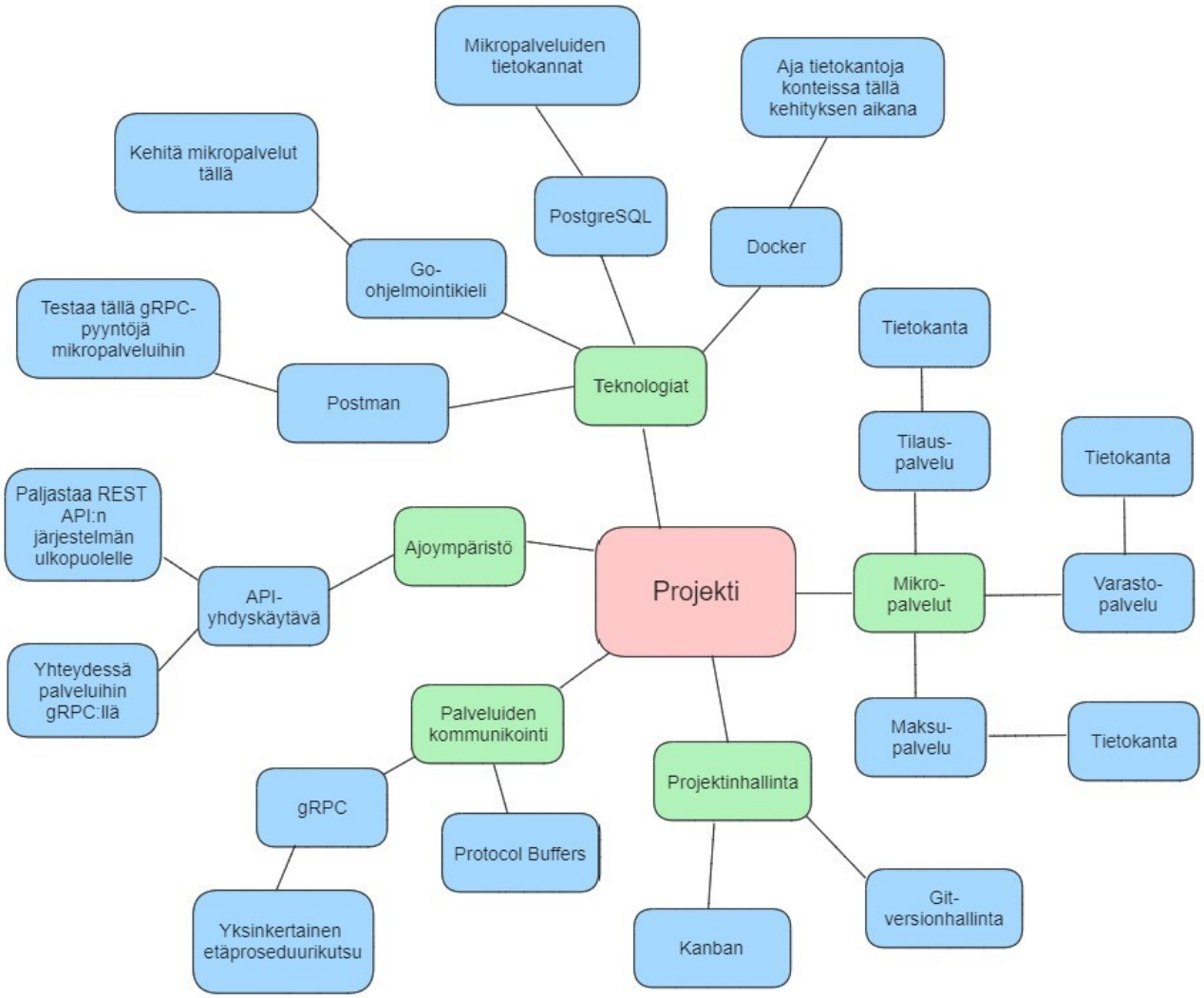
Liitteet

Liite 1. Tämän opinnäytetyön keskeiset käsitteet

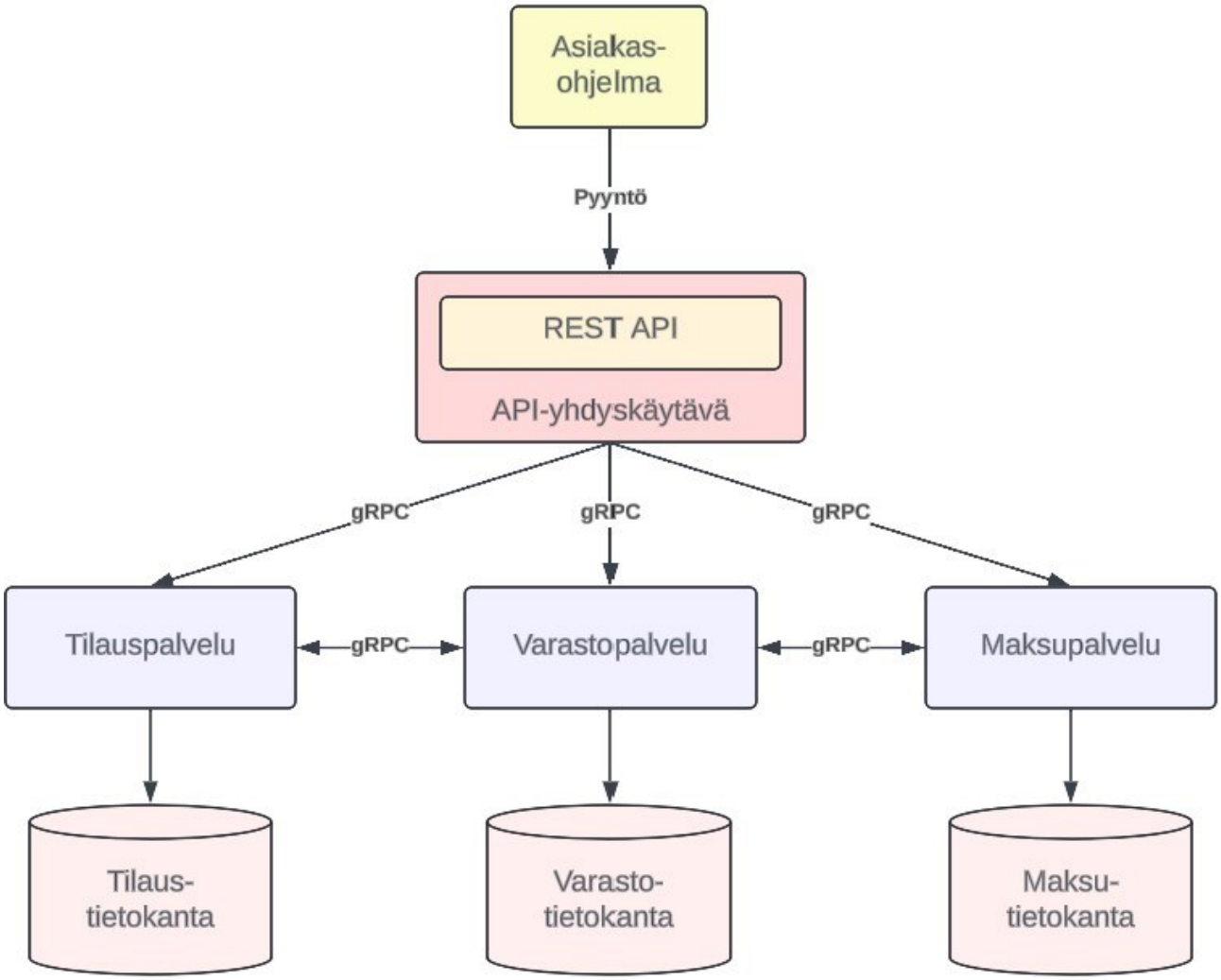
Käsite	Määritelmä
API	Application Programming Interface eli suomeksi ohjelmointirajapinta. Se on kahden ohjelmiston välissä oleva välittäjä, jonka avulla voidaan yhdistää kaksi tietokonetta tai tietokoneohjelmaa rajapinnan kautta (Acharya 22.6.2022).
Asiakasohjelmisto	Ohjelmisto, joka käyttää palvelinkoneella olevaa palvelinohjelmistoa. Se voi lähettää pyyntöjä palvelimelle, joka vastaa pyyntöihin lähettämällä asiakasohjelmistolle vastauksia.
Asynkroninen kommunikointi	Ohjelmistojen kommunikointitapa, jossa palvelinta kutsuva ohjelmisto ei odota, kunnes kutsuttava toiminto on suoritettu. Kutsuva ohjelmisto ei välttämättä edes välitä, suoritetaanko kutsuttavaa toimintoa loppuun asti. (Newman 2015, luku 4 alaluku Synchronous Versus Asynchronous.)
Binäärimuoto	Tapa esittää dataa käyttäen vain kahta merkkiä. Binäärimuodossa käytetään yleensä lukuja nolla ja yksi. (Nord Security 2024.)
gRPC	Avoimen lähdekoodin etäproseduurikutsuihin pohjautuva korkean suorituskyvyn kommunikointiprotokolla ja viitekehys.
Hajautettu järjestelmä	Eri tietokoneilla ajettavista prosesseista koostuva ohjelmistokokonaisuus, jossa prosessit kommunikoivat keskenään tietoverkon kautta.
HTTP	Lyhenne sanoista hypertext transfer protocol. Se on protokolla, jota tietokoneet ja palvelimet voivat käyttää tietojen pyytämiseen ja lähettämiseen verkon yli. (Cloudflare 2024.)
HTTP/2	Vuonna 2015 luotu uusi versio HTTP-protokollasta. HTTP/2 on paljon nopeampi ja tehokkaampi kuin alkuperäinen HTTP-protokolla. (Cloudflare 2024.)
Integraatio	Sovelluksen komponenttien yhdistämistä yhdeksi toimivaksi kokonaisuudeksi esimerkiksi API:en avulla.
Kommunikointiketju	Järjestys, jossa mikropalvelut lähettävät pyyntöjä toisillensa suorittaakseen jonkin useampaan mikropalveluun jakautuvan toiminnon.
Mikropalvelu	Mikropalveluarkkitehtuurissa käytettäviä käyttöönotettavia ohjelmistopalveluita, jotka toimivat irrallaan muista palveluista. (Acharya 22.6.2022.)
Minimaalinen toteutus	Asian toteuttamista siten, että siihen tehdään pääasiallisesti vain kaikki välttämättömät ja tarvittavat toimenpiteet asian toimiakseen jollakin tietyllä tavalla.
Palvelin	Palvelin voi olla palvelintietokone tai palvelinohjelmisto. Palvelintietokone on tietokone, joka ajaa palvelinohjelmistoja, joita asiakasohjelmistot voivat käyttää.

Käsite	Määritelmä
Prosessi	Tietokoneella ajettava ohjelma, jonka ohjelmakoodi on ladattu tietokoneen muistiin (Xu & Lam 15.12.2022, 0:15-0:35 min).
Protocol Buffers	gRPC:n oletuksena käyttämä datan sarjoittamismekanismi ja rajapinnan määritelmäkieli (Borysov & Gardiner 3.9.2021).
Protokolla	Joukko sääntöjä, jotka määrittelevät, miten ohjelmistot keskustelevat toistensa kanssa (Acharya 22.6.2022).
Rajapinnan määritelmäkieli	Kieli, jota käytetään ohjelmointirajapinnan määrittelemiseen. Sen avulla voidaan määritellä rajapinnan tarjoamat toiminnot, joita muut ohjelmistot voivat kutsua.
REST	Lyhenne sanoista Representational State Transfer. Se on suosituin arkkitehtuurityyli API:en rakentamiseen, jossa toiminnot suoritetaan resursseilla käyttäen standardoituja HTTP-metodeja. (The Postman Team 20.11.2023.)
RPC	Remote Procedure Call eli suomeksi etäproseduurikutsu. Etäproseduurikutsussa asiakasohjelma kutsuu funktiota palvelimella kuin funktio olisi paikallinen funktio asiakasohjelman puolella. Tyypillisesti kutsun mukana lähetetään dataa palvelimelle. (Amazon Web Services 2024b.)
Sarjoittaminen	Datan muuntamista muotoon, jossa se voidaan siirtää tai varastoida (Shuiskov 2022, luku 4 alaluku The basics of serialization).
Synkroninen kommunikointi	Ohjelmistojen kommunikointitapa, jossa palvelinta kutsuva ohjelmisto odottaa, kunnes kutsuttava toiminto on suoritettu (Newman 2015, luku 4 alaluku Synchronous Versus Asynchronous).
Säie	Prosessin sisällä oleva suoritusyksikkö, jossa ohjelmakoodia ajetaan. Jokaisella prosessilla on ainakin yksi säie. (Xu & Lam 15.12.2022, 1:25-1:35 min.)
Tyyppiturvallisuus	Järjestelmän ominaisuus, joka varmistaa, että ohjelmakoodin tietotyypeille voidaan tehdä vain toimintoja, jotka ovat sallittuja (Sandoval 21.2.2024).

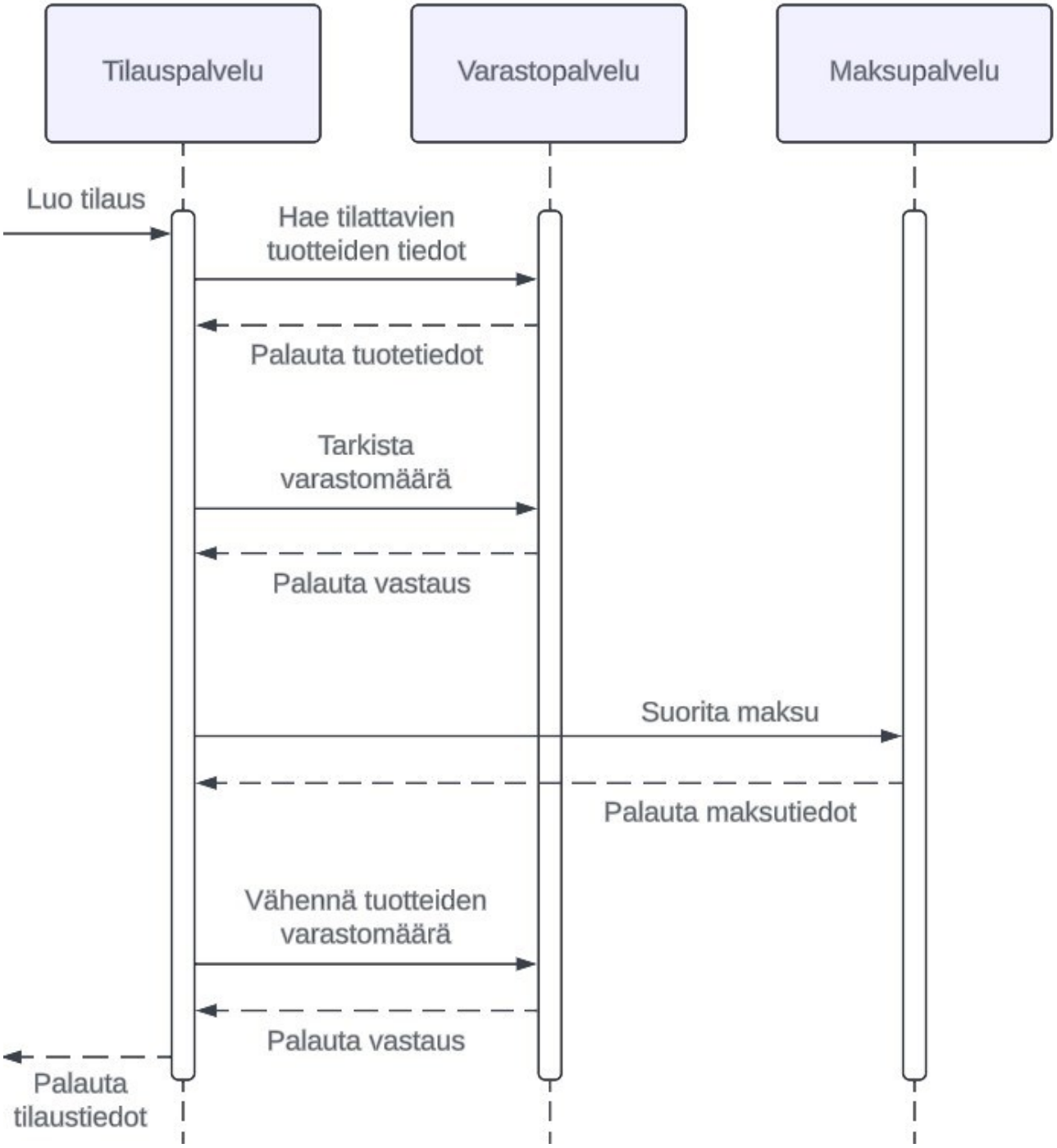
Liite 2. Projektin suunnitteluvaiheessa luotu ajatuskartta projektin sisällöstä



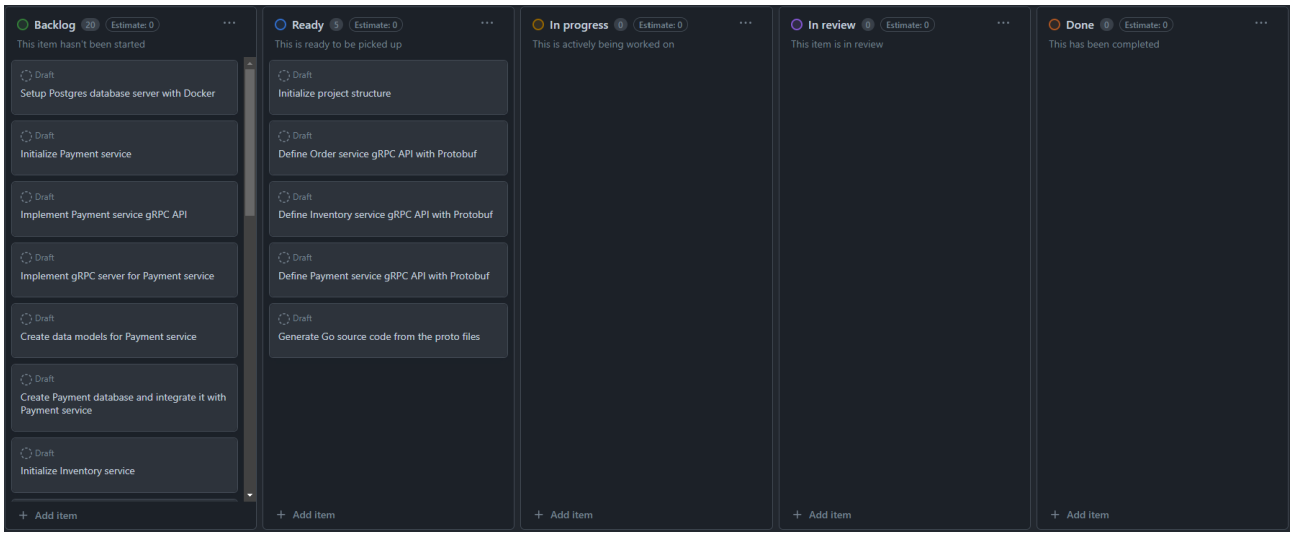
Liite 3. Projektin suunnitteluvaiheessa luotu arkkitehtuurikaavio



Liite 4. Projektin suunnitteluvaiheessa luotu tilaustoimintoa kuvaava sekvenssikaavio



Liite 5. Projektin Kanban-taulu kehityksen alussa



Liite 6. Mikropalveluiden gRPC API:en määritelmät Protocol Buffersilla

Tilauspalvelun prototiedosto.

```
// Order service gRPC API definitions
// v1.0.0

syntax = "proto3";
package orderpb;

option go_package = "/orderpb";

message OrderItem {
  // The code of the ordered product.
  string product_code = 1;
  // The Quantity of the ordered product.
  int32 quantity = 2;
}

message CreateOrderRequest {
  // The ID of the customer who created the order.
  string customer_id = 1;
  // The ordered products.
  repeated OrderItem order_items = 2;
}

message CreateOrderResponse {
  // The ID of the created order.
  string order_id = 1;
}

service OrderService {
  rpc CreateOrder(CreateOrderRequest) returns (CreateOrderResponse){}
}
```

Varastopalvelun prototiedosto.

```
// Inventory service gRPC API definitions
// v1.0.0

syntax = "proto3";
package inventorypb;

option go_package = "/inventorypb";

message ProductDetails {
  // The code that identifies the product.
  string product_code = 1;
  // The name of the product.
  string name = 2;
  // The description of the product.
  string description = 3;
  // The unit price of the product in cents.
  int32 unit_price_cents = 4;
}

message GetProductDetailsRequest {
  // The codes of the products whose details should be returned.
  repeated string product_codes = 1;
}

message GetProductDetailsResponse {
  // The returned product details.
  repeated ProductDetails product_details = 1;
}

message ProductQuantity {
  // The code of the product.·
  string product_code = 1;
  // The quantity of the product.
  int32 quantity = 2;
}

message ProductStock {
  // The code of the product.
  string product_code = 1;
  // The available quantity of the product in stock.
  int32 available_quantity = 2;
  // True if there are enough units of the product in stock. Otherwise false.
  bool is_available = 3;
}

message CheckProductStockQuantityRequest {
  // The products whose stock quantities should be checked.
  repeated ProductQuantity products = 1;
}

message CheckProductStockQuantityResponse {
  // The product stock quantity results.
  repeated ProductStock products = 1;
}

message ReduceProductStockQuantityRequest {
  // The products whose stock quantity should be reduced.
  repeated ProductQuantity products = 1;
}

message ReduceProductStockQuantityResponse {
  // The product stock quantity results after reducing the quantities.
  repeated ProductStock products = 1;
}

service InventoryService {
  rpc GetProductDetails(GetProductDetailsRequest) returns (GetProductDetailsResponse){}
  rpc CheckProductStockQuantity(CheckProductStockQuantityRequest) returns (CheckProductStockQuantityResponse){}
  rpc ReduceProductStockQuantity(ReduceProductStockQuantityRequest) returns (ReduceProductStockQuantityResponse){}
}
```

Maksupalvelun prototiedosto.

```
// Payment service gRPC API definitions
// v1.0.0

syntax = "proto3";
package paymentpb;

option go_package = "/paymentpb";

message CreatePaymentRequest {
  // The ID of the customer making the payment.
  string customer_id = 1;
  // The ID of the order that should be paid.
  string order_id = 2;
  // The total price to pay in cents.
  int32 total_price_cents = 3;
}

message CreatePaymentResponse {
  // The ID of the created payment.
  string payment_id = 1;
}

service PaymentService {
  rpc CreatePayment(CreatePaymentRequest) returns (CreatePaymentResponse){}
}
```

Liite 7. Skripti, jolla voi generoida prototiedostoista gRPC-koodit Go-kielelle

```
#!/bin/bash

# This script generates source codes from .proto files
# using Protobuf compiler protoc.

protoc --go_out=./gen/golang \
  --go-grpc_out=./gen/golang \
  def/**/*.proto
```

Liite 8. Maksupalvelun ohjelmistoriippuvuudet go.mod-tiedostossa

```
module github.com/hollowdll/go-grpc-microservices/services/payment

go 1.22

require (
	github.com/google/uuid v1.6.0
	github.com/hollowdll/grpc-microservices-proto/gen/golang/paymentpb v1.0.0
	github.com/spf13/viper v1.19.0
	google.golang.org/grpc v1.65.0
)

require (
	github.com/fsnotify/fsnotify v1.7.0 // indirect
	github.com/hashicorp/hcl v1.0.0 // indirect
	github.com/magiconair/properties v1.8.7 // indirect
	github.com/mitchellh/mapstructure v1.5.0 // indirect
	github.com/pelletier/go-toml/v2 v2.2.2 // indirect
	github.com/sagikazarmark/locafero v0.4.0 // indirect
	github.com/sagikazarmark/slog-shim v0.1.0 // indirect
	github.com/sourcegraph/conc v0.3.0 // indirect
	github.com/spf13/afero v1.11.0 // indirect
	github.com/spf13/cast v1.6.0 // indirect
	github.com/spf13/pflag v1.0.5 // indirect
	github.com/subosito/gotenv v1.6.0 // indirect
	go.uber.org/atomic v1.9.0 // indirect
	go.uber.org/multierr v1.9.0 // indirect
	golang.org/x/exp v0.0.0-20230905200255-921286631fa9 // indirect
	golang.org/x/net v0.27.0 // indirect
	golang.org/x/sys v0.22.0 // indirect
	golang.org/x/text v0.16.0 // indirect
	google.golang.org/genproto/googleapis/rpc v0.0.0-20240711142825-46eb208f015d // indirect
	google.golang.org/protobuf v1.34.2 // indirect
	gopkg.in/ini.v1 v1.67.0 // indirect
	gopkg.in/yaml.v3 v3.0.1 // indirect
)
```

Liite 9. Maksupalvelun gRPC-palvelin ja gRPC API

gRPC-palvelimen ajamisen lähdekoodi.

```
// Run runs the gRPC server and starts to listen for requests.
func (a *Adapter) Run() {
	log.Printf("initializing payment service gRPC server on port %d ...", a.config.GrpcPort)

	lis, err := net.Listen("tcp", fmt.Sprintf(":%d", a.config.GrpcPort))
	if err != nil {
		log.Fatalf("failed to listen on port %d: %v", a.config.GrpcPort, err)
	}

	grpcServer := grpc.NewServer()
	paymentpb.RegisterPaymentServiceServer(grpcServer, a)

	// this enables gRPC services to be tested with e.g. grpcurl
	if a.config.IsDevelopmentMode() {
		log.Println("development mode detected: enabling gRPC server reflection ...")
		reflection.Register(grpcServer)
	}

	log.Printf("starting payment service gRPC server ...")
	log.Printf("payment service gRPC server listening at %v", lis.Addr())

	if err = grpcServer.Serve(lis); err != nil {
		log.Fatalf("failed to serve gRPC at %v", lis.Addr())
	}
}
```

Ohjelmoidun CreatePayment-nimisen etäproseduurikutsun lähdekoodi.

```
func (a *Adapter) CreatePayment(ctx context.Context, req *paymentpb.CreatePaymentRequest) (res *paymentpb.CreatePayment
Response, err error) {
	log.Printf("call RPC %s: request = %v", createPaymentPrefix, req)
	defer func() {
		if err != nil {
			log.Printf("RPC %s failed: request = %v; error = %v", createPaymentPrefix, req, err)
		} else {
			log.Printf("RPC %s success: request = %v; response = %v", createPaymentPrefix, req, res)
		}
	}()

	newPayment := domain.NewPayment(req.CustomerId, req.OrderId, req.TotalPriceCents)
	result, err := a.api.Charge(ctx, newPayment)
	if err != nil {
		return nil, status.Error(codes.Internal, fmt.Sprintf("failed to charge: %v", err))
	}

	return &paymentpb.CreatePaymentResponse{PaymentId: result.ID}, nil
}
```

Liite 10. Tilauspalvelun gRPC-asiakas

gRPC-asiakkaan luonnin lähdekoodi.

```
func NewAdapter(cfg *config.Config) (*Adapter, error) {
	log.Printf("creating gRPC client for payment service ...")

	var opts []grpc.DialOption
	opts = append(opts, grpc.WithTransportCredentials(insecure.NewCredentials()))
	address := fmt.Sprintf("%s:%d", cfg.PaymentServiceHost, cfg.PaymentServicePort)

	log.Printf("using endpoint %s", address)
	conn, err := grpc.NewClient(address, opts...)
	if err != nil {
		return nil, err
	}
	client := paymentpb.NewPaymentServiceClient(conn)

	return &Adapter{
		payment: client,
		conn:    conn,
	}, nil
}
```

Generoidun gRPC-ohjelmakoodin käyttäminen gRPC-pyyntöjen lähettämiseksi maksupalveluun.

```
func (a *Adapter) CreatePayment(ctx context.Context, order *domain.Order, totalPriceCents int32) error {
	_, err := a.payment.CreatePayment(ctx, &paymentpb.CreatePaymentRequest{
		CustomerId:      order.CustomerID,
		OrderId:         order.ID,
		TotalPriceCents: totalPriceCents,
	})
	return err
}
```

Liite 11. Tilauspalvelun tilaustoiminnon liiketoimintalogiikka

```
func (a *Application) CreateOrder(ctx context.Context, order *domain.Order) (*domain.Order, error) {
	// 1. get product prices
	var productCodes = []string{}
	for _, orderItem := range order.OrderItems {
		productCodes = append(productCodes, orderItem.ProductCode)
	}

	productPrices, err := a.inventory.GetProductPrices(ctx, productCodes)
	if err != nil {
		return nil, ordererrors.ErrGetProductPrices
	}

	// 2. check product stock quantities
	productStocks, err := a.inventory.CheckProductStockQuantities(ctx, order.OrderItems)
	if err != nil {
		return nil, ordererrors.ErrCheckProductStockQuantities
	}
	for _, productStock := range productStocks {
		if !productStock.IsAvailable {
			return nil, ordererrors.ErrNotEnoughProducts
		}
	}

	// 3. make payment
	totalPriceCents := sumPrices(productPrices, order.OrderItems)
	err = a.payment.CreatePayment(ctx, order, totalPriceCents)
	if err != nil {
		return nil, ordererrors.ErrCreatePayment
	}

	// 4. reduce product stock quantity
	err = a.inventory.ReduceProductStockQuantities(ctx, order.OrderItems)
	if err != nil {
		return nil, ordererrors.ErrReduceProductStockQuantities
	}

	return order, nil
}
```

Liite 12. Ohjelmoitu CreateOrder-niminen etäproseduurikutsu tilauspalvelun gRPC API:ssa

```
func (a *Adapter) CreateOrder(ctx context.Context, req *orderpb.CreateOrderRequest) (res *orderpb.CreateOrderResponse, err error) {
	log.Printf("call RPC %s: request = %v", createOrderName, req)
	defer func() {
		if err != nil {
			log.Printf("RPC %s failed: request = %v; error = %v", createOrderName, req, err)
		} else {
			log.Printf("RPC %s success: request = %v; response = %v", createOrderName, req, res)
		}
	}()

	// use timeout so the order operation won't wait forever
	ctx, cancel := context.WithTimeout(ctx, createOrderTimeout)
	defer cancel()

	var orderItems []*domain.OrderItem
	for _, orderItem := range req.OrderItems {
		orderItems = append(orderItems, &domain.OrderItem{
			ProductCode: orderItem.ProductCode,
			Quantity:    orderItem.Quantity,
		})
	}
	newOrder, err := domain.NewOrder(req.CustomerId, orderItems)
	if err != nil {
		log.Printf("RPC %s: failed to create new order: %v", createOrderName, err)
		return nil, status.Error(codes.Internal, "order creation failed")
	}

	result, err := a.api.CreateOrder(ctx, newOrder)
	if err != nil {
		return nil, status.Error(codes.Internal, fmt.Sprintf("order creation failed: %v", err))
	}

	return &orderpb.CreateOrderResponse{OrderId: result.ID}, nil
}
```

Liite 13. Onnistuneen tilauspalvelun tilaustoiminnon testaus grpcurl-ohjelmalla

Pyynnön data JSON-muodossa.

```
juuso@machine:~/git/go-grpc-microservices/services/order/tests/testdata/CreateOrder$ cat request_success.json
{
  "customer_id": "customer-123",
  "order_items": [
    {
      "product_code": "0190e8c4-258e-767f-94a7-b5183aea900f",
      "quantity": 2
    },
    {
      "product_code": "0190e8c4-258e-7693-ab70-c5f2da6739db",
      "quantity": 1
    }
  ]
}
```

Pyynnön lähettäminen ja sen vastaus kutsumalla CreateOrder-nimistä etäproseduurikutsua.

```
juuso@machine:~/git/go-grpc-microservices/services/order/tests/testdata/CreateOrder$ grpcurl -plaintext -d @ localhost:9002
orderpb.OrderService/CreateOrder < request_success.json
{
  "orderId": "0190e924-6545-70a6-82d4-e6ce296d2cee"
}
```

Liite 14. Epäonnistuneen tilauspalvelun tilaustoiminnon testaus grpcurl-ohjelmalla

Pyynnön data JSON-muodossa.

```
juuso@machine:~/git/go-grpc-microservices/services/order/tests/testdata/CreateOrder$ cat request_out_of_stock.json
{
  "customer_id": "customer-123",
  "order_items": [
    {
      "product_code": "0190e8c4-258e-7697-8bde-98508cde1a97",
      "quantity": 5
    }
  ]
}
```

Pyynnön lähettäminen ja sen virhevastaus kutsumalla CreateOrder-nimistä etäproseduurikutsua.

```
juuso@machine:~/git/go-grpc-microservices/services/order/tests/testdata/CreateOrder$ grpcurl -plaintext -d @ localhost:9002
orderpb.OrderService/CreateOrder < request_out_of_stock.json
ERROR:
  Code: Internal
  Message: order creation failed: not enough ordered products in stock
```

Virheen sattuminen varastopalvelun ollessa alhaalla.

```
juuso@machine:~/git/go-grpc-microservices/services/order/tests/testdata/CreateOrder$ grpcurl -plaintext -d @ localhost:9002
orderpb.OrderService/CreateOrder < request_success.json
ERROR:
  Code: Internal
  Message: order creation failed: cannot get prices of the ordered products
```

Virheen sattuminen maksupalvelun ollessa alhaalla.

```
juuso@machine:~/git/go-grpc-microservices/services/order/tests/testdata/CreateOrder$ grpcurl -plaintext -d @ localhost:9002
orderpb.OrderService/CreateOrder < request_success.json
ERROR:
  Code: Internal
  Message: order creation failed: failed to process payment
```