Kertakirjautuminen mikropalveluna

Ville Koskela

Haaga-Helia ammattikorkeakoulu

Tradenomi

Opinnäytetyö

2022

Tiivistelmä

Tekijä(t)
Ville Koskela

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
Kertakirjautuminen mikropalveluna

Sivumäärä
36

Käyttäjän tunnistaminen ja valtuuttaminen on tärkeä osa monia verkkopalveluita. Käyttäjän näkökulmasta voi kuitenkin olla haastavaa muistaa lukuisia eri tunnuksia ja kirjautua erikseen jokaiseen palveluun. Tässä opinnäytetyössä on kuvailtu ratkaisun etsimistä esitettyyn ristiriitaan.

Opinnäytetyössä keskityttiin kertakirjautumispalvelun kuvailuun ja keskeisenä tietoperustana toimi OAuth 2.0 kehyksen dokumentaatio. Kertakirjautumisen lisäksi opinnäytetyössä on käsitelty mikropalveluita ja mikropalveluarkkitehtuurin tuomia etuja, joihin on todettu kuuluvan mm. skaalattavuus, joustavampi kehitystyö ja ketterämpi julkaiseminen.

Alkuperäisenä ideana oli toteuttaa yksinkertainen kirjautumisrajapinta käyttäjäntunnistusta vaativille palveluille, mutta toteutustavassa havaittiin merkittäviä puutteita. Ratkaisua lähdettiin etsimään OAuth 2.0 kehyksestä, jonka keskeisenä etuna nähtiin, ettei käyttäjän tarvitse syöttää tunnistetietojaan kuin kerran saadakseen pääsyn lukuisiin palveluihin. Kehitysprosessin aikana havaittiin myös tarve pilkkoa kasvanut toteutus mikropalveluiksi, joiden avulla kehitystyöhön ja julkaisemiseen haluttiin saada joustavuutta.

Toteutetussa palvelussa haluttiin korostaa käyttäjän oikeutta hallita omia tietojaan. Haluttiin antaa käyttäjälle mahdollisimman laaja valta päättää, mitä tietoja hänestä voidaan tunnistautumisen yhteydessä luovuttaa ja mille taholle. Käyttäjälle tarjottiin myös mahdollisuus tulla unohdetuksi yhdellä pyynnöllä kaikista toteutettua kertakirjautumispalvelua käyttävistä palveluista.

Lopullisessa tuotteessa havaittiin runsasta potentiaalia, jonka voisi valjastaa käyttöön pienen jatkokehityksen avulla. Sen katsottiin voivan tarjota sulavan käyttäjähallinnan laajoihin palvelukokonaisuuksiin, joissa käyttäjä siirtyy vaivattomasti palvelusta toiseen. Toteutetusta tuotteesta olisi melko pienin muutoksin mahdollista toteuttaa ratkaisu käyttäjätietojen siirtoon käyttäjän mukana ilman, että tietojensiirto näkyy käyttäjälle millään tavalla.

Asiasanat
OAuth, mikropalvelu, kertakirjautuminen, todentaminen, valtuuttaminen

Sisällys

1 Johdanto

Tässä opinnäytetyössä kuvaillaan kertakirjautumispalvelun toteuttamista mikropalveluna. Johdannon ensimmäisessä alaluvussa kuvaillaan työn tavoitteita. Toisessa alaluvussa pyritään avaamaan työn taustoja ja valmiin tuotteen käyttökohteita.

1.1 Opinnäytetyön tavoitteet

Monien verkkopalveluiden käyttö edellyttää jonkinlaista tunnistautumista käyttäjältä. Usein tämä edellyttää tunnusten luomista palveluun ja jatkossa käyttäjä kirjautuu palveluun luomallaan käyttäjätunnuksella ja salasanalla. Seurauksena tästä on, että käyttäjillä kertyy lukuisia tunnuksia ja salasanoja muistettavaksi. Ratkaisuna tähän toimivat kertakirjautumispalvelut (single sign-on tai lyhyesti SSO). Tällaista kertakirjautumista tarjoavat mm. Google ja Facebook. Niiden käyttö kuitenkin edellyttää tiliä kyseisissä palveluissa.

Tässä opinnäytetyössä toteutetaan yksinkertainen ja kevyt keskitetty käyttäjähallinta erilaisiin webpohjaisiin palveluihin siten, että se mahdollistaa useiden palveluiden käytön samalla tunnuksella. Työssä on tarkoitus selvittää, miten seuraavat asiat ovat järkevintä toteuttaa:

1. Miten välittää luotettavasti ja turvallisesti tieto onnistuneesta kirjautumisesta käyttäjän tunnistamista vaativalle palvelulle?
2. Miten tarjota käyttäjälle täysi valta päättää, mitä tietoja hänestä luovutetaan eteenpäin ja minne näitä tietoja saa luovuttaa?
3. Mitä etuja saavutetaan, jos ratkaisu pilkotaan moderneiksi mikropalveluiksi?

Kyseessä on toiminnallinen opinnäytetyö, jonka tuotoksena on toimiva ohjelmistotuote. Tuotteen toteutuskielenä käytetään Typescriptiä, jonka tarjoama tyypitys tekee jatkokehityksen ja ylläpidon helpommaksi. Lisäksi kielivalinta mahdollistaa NPM-pakettien käytön sellaisiin toiminnallisuuksiin, joita ei ole mielekästä kirjoittaa itse. Ulkoiset riippuvuudet pyritään kuitenkin pitämään minimaalisina, jotta sovellus pysyisi kompaktina ja kevyenä ja tarjoaisi mahdollisimman paljon mahdollisuuksia oppimiseen.

Opinnäytetyössä kuvaillaan alussa kirjautumispalvelun kannalta keskeisiä käsitteitä, joita myöhemmin käytetään toteutusprosessia kuvailtaessa. Keskeisimpinä käsitteinä esille nousevat sellaiset käsitteet, jotka liittyvät tutkimuskysymyksiin ja siten liittyvät olennaisesti haasteisiin kirjautumispalvelun toteuttamisessa.

1.2 Opinnäytetyön taustat

Opiskeluaikana toteutetaan usein erilaisia ohjelmistoprojekteja ja monesti ohjelmistotuotteet tarvitsevat jonkinlaisen käyttäjähallinnan. Tämä voi osoittautua kuormittavaksi, kun useissa projekteissa päädytään tekemään uudestaan ja uudestaan samaa asiaa: käyttäjänhallintaa. Lisäksi tietoturvan näkökulmasta ongelmallista on, että salasanoja tallennetaan eri palveluiden tietokantoihin. Ongelman ratkaisuksi lähdettiin selvittämään keskitetyn kirjautumispalvelun toteuttamista, minkä myötä salasanoja ei jatkossa olisi kuin kyseisen palvelun tietokannassa ja samat tunnukset toimisivat kaikkiin projekteihin. Erityisesti hyötyä haetaan siitä, ettei uutta palvelua toteuttaessa menisi enää aikaa käyttäjähallinnan kirjoittamiseen, vaan tätä asiaa varten voisi esimerkiksi kirjoittaa valmiin lisäosan, jonka liittää aina uuteen projektiin.

Vaikka opinnäytetyö tulee yksityiseen käyttöön, sen ajankohtaisuutta ja tarpeellisuutta ei tule vähätellä. Internetin aikakaudella yksi keskeisiä tarpeita on tunnistaa käyttäjä ja varmistaa, mitä käyttäjällä on lupa tehdä tunnistautumisen jälkeen. Tämä kaikki tulisi lisäksi toteuttaa siten, että käyttäjäkokemus olisi mahdollisimman mukava tai vaarana on käyttäjän siirtyminen toiseen palveluun.

Opinnäytetyöprosessin aikana aiheen tuntemus lisääntyi ja lopullinen tuote ylitti alkuperäiset tavoitteet. Toteutettu tuote tarjoaa jatkokehitysmahdollisuuksia, joiden avulla siitä voisi kehittää hyvän tuotteen yritykselle, jolla on useita toisistaan erillisiä web-pohjaisia palveluita, mutta joiden välillä käyttäjän halutaan pääsevän siirtymään joustavasti ja vaivattomasti. Arkkitehtuuri muuntui tuotetta kehitettäessä ja lopullinen versio on pilkottu useaksi mikropalveluksi siten, että se tarjoaa myös skaalausmahdollisuuksia käyttäjämäärien kasvaessa.

2 Mikropalveluarkkitehtuuri

Tässä luvussa pyritään ensin avaamaan mikropalvelun käsitettä ja määrittelemään se, mitä mikropalvelun käsitteellä myöhemmin tarkoitetaan. Mikropalveluarkkitehtuurin tarjoamia etuja ja haasteita tarkastellaan niin sovelluskehittäjän näkökulmasta kuin myös skaalaamisen näkökulmasta. Lopuksi pohditaan sitä, milloin on mielekästä irrottaa toiminnallisuus omaksi mikropalvelukseen.

2.1 Mikä on mikropalvelu?

Mikropalvelun käsitteelle on vaikea löytää täysin yksiselitteistä määritelmää. Atchison (2022) ei pidä koko mikropalvelun käsitteestä, koska se antaa hänen mukaansa kuvan pienistä palveluista. Atchinsin mukaan tällainen mikropalvelu voi kuitenkin olla verrattain suuri joskin rajattu palvelu. Taibi, Lenarduzzi, Pahl ja Janes (2017) pitävätkin mikropalvelun keskeisenä kriteerinä sitä, että palvelulla on yksi selkeästi rajattu tehtävä, jota se hoitaa itsenäisesti. Myös Fowler ja Lewis (2014) tuovat tätä näkökulmaa esiin todetessaan, ettei mikropalvelulle ole olemassa selkeää yksiselitteistä määritelmää. Fowlerin ja Lewisin mukaan keskeistä kuitenkin on palvelun hajottaminen useisiin pienempiin osiin. Tämän voi ymmärtää siten, että suuri palvelukokonaisuus hajotetaan pienempiin itsenäisiin yksiköihin, jotka toteuttavat itsenäisesti omaa rajattua tehtäväänsä.

Mikropalveluarkkitehtuurista puhuttaessa se usein nähdään vastakohtana monoliittiselle sovellukselle, joka on Atchisonin (2022) mukaan ollut perinteisesti yleisin tapa tuottaa ohjelmistoja. Monoliittisen sovelluksen määritteleminen jätettäköön tässä kohtaa tekemättä, mutta siinä kaikki halutut toiminnot ovat sisäänrakennettuna yhden suoritettavan sovelluksen sisään (Fowler ja Lewis, 2014). Tällöin voitaneen tehdä oletus, että sovellusta voidaan suorittaa suoraan käyttöjärjestelmän päällä itsenäisenä sovelluksena. Atchinson (2022) huomauttaakin, että mikropalvelu voi olla pelkkä funktio, jota suoritetaan jonkin pilvipalvelun tarjoamassa ympäristössä. Tällöin on syytä huomata, ettei funktio yksinään kykenisi palvelemaan ketään, vaan se tarvitsee pilvipalvelun kutsumaan itseään ja huolehtimaan suorituksesta - sitä ei siis kutsu suoraan käyttöjärjestelmä! Atchison (2022) toteaakin, että mikropalveluarkkitehtuurissa osan palvelun komponenteista valmistaa pilvipalvelun toteuttajan insinöörit eikä palvelua kehittävät insinöörit.

Toisenlaisen perspektiivin mikropalveluiden määrittelyyn tarjoaa Nadareishvili, Mitra, McLarty ja Amundsen (2016, 9) toteamalla, että mikropalveluissa ei keskeistä ole teknologia tai työkalut vaan kyky mukautua. Heille mikropalveluarkkitehtuurissa on keskeistä mukautuvuus ja tällä he tarkoittavat sitä, että mikropalvelut mahdollistavat muutosten päätymisen entistä nopeammin työntekijöiden koneilta tuotantoympäristöön loppukäyttäjien saataville. Määritelmä ei sinänsä tuota ristiriitaa aiemmin kuvattujen teknisempien määritelmien kanssa. Tässä työssä katsotaan kuitenkin mikropalveluiden olevan yksi ohjelmistoarkkitehtuuri, joka on teknologinen ratkaisu ja teknologista ratkaisua ei

tulisi määritellä sen pohjalta, mitä se mahdollistaa. Ei nimittäin ole taetta, etteikö joku toinen teknologia voisi tarjota samoja mahdollisuuksia liittymättä mitenkään mikropalveluihin. Sen sijaan voi tehdä tulkinnan, että Nadareishvili ym. (2016, 9) piirtävät yhtäläisyysmerkkiä mikropalveluiden ja ketterän kehitysmallin välille.

Tässä työssä mikropalvelu ja mikropalveluarkkitehtuuri määritellään nimenomaan siten, että suurempi kokonaisuus on pilkottu pienempiin osiin ja näillä pienemmillä osilla on selkeästi rajatut omat tehtävänsä, joista ne huolehtivat itsenäisesti. Tämä nähdään toimivana määritelmänä, koska se ei sinänsä ota kantaa teknologiaan, kuten kontittamiseen tai käytettyihin työskentelymenetelmiin, kuten erilaiset ketterät menetelmät. Lisäksi se antaa perustelun mikro-etuliitteelle, koska kyse on nimenomaan suuremman ohjelmiston paloittelusta, vaikka joku yksittäinen palanen olisikin suurempi kuin toinen. Tällainen määritelmä myös kuvaa arkkitehtuuria itseään, eikä sitä, mitä kyseinen arkkitehtuuri mahdollistaa.

Mikropalvelun määritelmä nojaa vahvasti itsenäisyyden käsitteeseen. Atchison (2020) mukaan itsenäiseksi palveluksi voidaan kutsua sellaista palvelua, joka täyttää viisi erityistä ehtoa. Näistä ehdoista ensimmäinen koskee koodia ja toinen dataa ja ehtojen mukaan näiden molempien tulee olla täysin itsenäisiä, eli mikropalvelun ei tule jakaa koodiaan toisen palvelun kanssa. Samoin sen datan tulee olla sen omaa, jolloin se ei käytä suoraan muiden palveluiden dataa eikä muut palvelu käytä suoraan sen dataa. Seuraavat kaksi ehtoa puolestaan koskevat rajapintoja: mikropalvelun tulee tarjota toimintojaan julkisten rajapintojensa kautta muille palveluille ja vastaavasti sen tulee käyttää muiden palveluiden tarjoamia toimintoja näiden julkisten rajapintojen kautta. Viimeinen ehto liittyy hallinnointiin ja sen mukaan mikropalvelulla voi olla vain yksi kehittäjätiimi. (Atchison 2020.)

2.2 Mikropalvelut kehittäjän näkökulmasta

Taibi, Lenarduzzi, Pahl ja Janes (2017) ovat sitä mieltä, että monoliittisen sovelluksen kehitys ja käyttöönotto voi olla helpompaa etenkin kokemattomalle kehittäjälle. Fowler ja Lewis (2014) tukevat tätä näkemystä toteamalla, että yksittäiselle sovellukselle on helpompi rakentaa tuotantoympäristö testi- ja julkaisuautomaatioineen. Kuitenkin Taibi ym. (2017) toteavat, että kasvaessaan monoliittinen sovellus muuttuu haastavaksi kehittää, koska testattavaa on paljon ja pienetkin muutokset vaativat suurien testikokonaisuuksien ajamista. Haastavuutta lisää myös se, että jokainen ohjelmoija työskentelee saman koodikannan parissa. Atchisonin (2022) mukaan mikropalveluarkkitehtuuri helpottaa tätä pilkkomalla luontevasti koodin pienempiin itsenäisiin osiin, jolloin vain kyseisen osan parissa työskentelevien tarvitsee ymmärtää ja tuntea koodin sisäinen toimintalogiikka.

Mikropalvelut tarjoavatkin kehittäjälle monia etuja monoliittiseen toteutukseen nähden. Koska jokainen mikropalvelu on itsenäinen komponentti, joka keskustelee toisten komponenttien kanssa julkisten rajapintojen kautta, syntyy eri toimintojen välille erittäin vahva kapselointi (Fowler ja Lewis, 2014). Tämä tarkoittaa sitä, ettei ohjelmoijan ole mahdollista rakentaa eri komponenttien välillä riippuvuuksia, jotka voisivat myöhemmin aiheuttaa ongelmia. Atchison (2022) täydentää näkemystä toteamalla, että monoliittisessa sovelluksessa kaikki ohjelmoijat työskentelevät saman koodikannan parissa, jolloin on suurempi todennäköisyys syntyä toistensa kanssa ristiriidassa olevia muutoksia. Juuri tähän aiemmin mainittu luonnollinen kapselointi voi tuoda avun. Taibi, Lenarduzzi, Pahl ja Janes (2017) huomauttavat myös, että tämä riippumattomuus mahdollistaa lisäksi eri ohjelmointikielien käytön eri mikropalveluiden toteuttamisessa, jolloin eri tehtäviin on mahdollista valita kyseiseen tehtävään parhaiten sopiva kieli. Näin mikropalveluarkkitehtuurissa eri komponentteja voi kehittää eri ohjelmoijat käyttäen itselleen tuttuja ja työstämäänsä komponenttiin parhaiten soveltuvia työkaluja.

Vaikka aiemmin totesimme monoliittisen sovelluksen testaamisen muuttuvan ajan kanssa hitaaksi valtavan testimassan takia, tarjoaa mikropalveluarkkitehtuurikin omat haasteensa testaamisen suhteen. Koska palvelut ovat pieniä itsenäisiä komponentteja, on niiden itsenäinen testaaminen kohtalaisen kevyttä, mutta integraatiotestaaminen muuttuu haastavammaksi. Fowler ja Lewis (2014) ovat kuitenkin sitä mieltä, että infrastruktuurin automatisointi on kehittynyt valtavasti ja nykyiset pilvipalvelut tarjoavat laadukkaita työkaluja, joiden avulla automatisoida eri testitasot tuotantoputkessa.

Kehittäjän näkökulmasta myös Nadareishvili, Mitra, McLarty ja Amundsen (2016, 9) havaitsema yhteys mikropalveluiden ja ketterien menetelmien välillä on merkityksellinen. Jos mikropalvelut todella mahdollistavat koodin nopeamman pääsyn tuotantoon, helpottavat ne myös ketterien ohjelmistokehityksen menetelmien käyttöönottoa. Tämä tarkoittaa sitä, että muutoksia voidaan saattaa nopealla syklillä tuotantoon ja kehitystyötä voidaan ohjata todellisella loppukäyttäjiltä saadulla palautteella ja näin voidaan olettaa syntyvän tuotteen palvelevan entistä paremmin sitä tarkoitusta, johon sitä tuotetaan. Nadareishvili, Mitra, McLarty ja Amundsen (2016, 9-10) kuitenkin huomauttavat, että nopeaan julkaisutahtiin liittyy myös riskejä. He korostavat sitä, että jokainen palvelun päivitys voi myös mahdollisesti sisältää virheitä ja näin ollen nopeassa syklissä tapahtuvat päivitykset palveluun lisäävät riskiä, että jokin päivitys tuo mukanaan palvelun normaalia toimintaa haittaavan vian (Nadareishvili, Mitra, McLarty ja Amundsen 2016, 9-10).

2.3 Palvelun skaalaaminen

Palvelun käyttäjäkunnan kasvaessa, voi ilmetä tarve skaalata palvelua vastaamaan kasvanutta kysyntää. Ennen kuin käsittelemme mikropalveluiden suhdetta skaalaamiseen, on todettava, että

skaalaaminen voidaan tehdä kahdella eri tavalla: horisontaalisesti tai vertikaalisesti. Vertikaalinen skaalaaminen on hyvin yksinkertaista: lisätään nykyisen laitteiston suorituskykyä, jolloin voidaan ajatella sen suorittavan ohjelmistoa nopeammin. Horisontaalisessa skaalaamisessa puolestaan lisätään nykyisen laitteiston rinnalle uusia yksiköitä, jolloin yksiköt voivat jakaa kuormaa keskenään. (Isaacson 2014.) Ymmärrettävästi on rajansa, miten nopean suorittimen ja miten paljon muistia yhteen tietokoneeseen voi laittaa ja tämä asettaa rajat sille, miten pitkälle vertikaalisella skaalaamisella voidaan päästä. Sen sijaan rinnakkaisten laitteistojen määrälle ei ole mitään rajaa, joten horisontaalisella skaalaamisella ei ole tällaista rajoitetta.

Fowler ja Lewis (2014) ovat sitä mieltä, että monoliittistakin sovellusta voi skaalata horisontaalisesti ottamalla käyttöön kuormantasaajan ja suorittamalla useampaa instanssia monoliittisesta sovelluksesta tasaajan takana. Tämä ei kuitenkaan välttämättä ole tehokkain tapa skaalaamiseen, sillä tällöin joudutaan skaalaamaan kaikkia palvelun osia riippumatta siitä, mitkä osat tosiasiassa eniten kuormittuvat. Newman (2019) toteaakin, että mikropalveluarkkitehtuuri mahdollistaa kustannustehokkaan skaalaamisen, koska voidaan skaalata horisontaalisesti vain niitä osia, jotka hitautta palvelussa aiheuttavat.

2.4 Huomioitavaa pilkottaessa

Aiemmin päätetyn määritelmän mukaan mikropalveluarkkitehtuurissa on kyse suuren kokonaisuuden hajottamisesta pieniksi itsenäisiksi palveluiksi. On kuitenkin syytä pohtia, mitkä asiat kannattaa pilkkoa omiksi palveluikseen ja milloin tuo pilkkominen ei ole mielekästä. Atchisonin (2020) mukaan tähän ei ole yhtä oikeaa ratkaisua, mutta hän näkee trendin olevan kohti aina vain pienempiä ja pienempiä palveluita.

Pohdittaessa pilkkomisen mahdollisuutta, kannattaa huomio pitää aiemmin mikropalvelun määritelmän yhteydessä kuvatuissa Atchisonin (2020) viidessä ehdossa itsenäiselle palvelulle. Näistä on syytä nostaa esille erityisesti datan merkitys. On helppo luoda repositorio, johon perustaa uuden koodikannan ja määrää tietyn kehittäjätiimin kehitystyöhön. Jos palvelu julkaistaan omana instanssinaan, ei kommunikaatioonkaan jää käytännössä ulkoisten rajapintojen lisäksi monia vaihtoehtoja. Sen sijaan datan erottaminen on tärkeää huomata, sillä sen kohdalla on helppo syyllistyä esimerkiksi käyttämään samaa tietokantaa toisen palvelun kanssa. Tällöin myös saatetaan helposti rikkoa ehtoa siitä, miten kommunikointi toisten palveluiden kanssa tulisi tapahtua.

Myös Fowler ja Lewis (2014) korostavat sitä, että jokaisella mikropalvelulla tulee olla oma datansa, jota vain kyseinen mikropalvelu pystyy käsittelemään. He näkevät tässä senkin edun, että jokainen palvelu voi valita itselleen sopivimman tavan tiedon säilyttämiseen toisin kuin monoliittisessa sovelluksessa, jossa yleensä käytetään yhtä tietokantaa kaiken pysyvän datan tallentamiseen (Fowler &

Lewis 2014). Toisin sanoen kehittäjät voivat valita relaatiotietokannan silloin, kun tämä on toteutuksen kannalta perusteltua tai toisaalta dokumenttitietokannan silloin, kun se on toimivampi ratkaisu palvelun näkökulmasta. Erityisesti tämä jousto on huomionarvoista kehitystyön ja ylläpidon näkökulmasta.

3 Kertakirjautuminen

Tässä luvussa käsitellään keskeisenä käsitteenä OAuth 2.0 kehystä, jonka määrittelee RFC 6749. RFC:t ovat asiakirjoja, joissa määritellään Internetiin liittyviä toimintoja ja osa näistä asiakirjoista on Internet Engineering Task Forcen hyväksymiä standardeja (NIIT 2002). RFC 6749 on hyväksytty Internetin standardi (RFC Editor s.a.). Luvun keskeiset käsitteet on kuvattu taulukossa 1.

Taulukko 1. Luvun 3 keskeiset käsitteet

Käsite	Selitys
CORS	Lyhenne tulee sanoista Cross Origin Resource Sharing. Se tarkoittaa sitä, että verkkosivu hakee tai lähettää tietoa toiseen verkko-osoitteeseen, kuin mistä verkkosivu itse on ladattu. (MDN Web Docs. 2022a.)
SPA-sovellus	SPA on lyhenne sanoista single page application. Kyseessä on verkkosivu, joka ladataan kerran selaimeen ja tämän jälkeen sen eri toiminnallisuudet ovat käytettävissä ilman sivun uudelleenlataamista.
Token	Tässä yhteydessä tokenilla tarkoitetaan JSON web tokenia, joka on RFC 7519:n määrittelemä tiivis tietorakenne. Se voidaan kuljettaa turvallisesti http-pyynnön otsaketiedoissa osoiterivin sijaan. (Internet Engineering Task Force 2015a.)

3.1 Todennus ja valtuutus

Käyttäjän kirjautuminen sisään palveluun sisältää kaksi vaihetta. Näistä ensimmäisessä käyttäjä todennetaan (authentication) ja toisessa tunnistettu käyttäjä valtuutetaan (authorization). Käsitteet on helppo sotkea keskenään, mutta niillä on merkitysero. Todentamisessa on kyse siitä, että tunnistetaan kuka on käyttämässä palvelua. Valtuuttaminen tapahtuu aina todentamisen jälkeen ja siinä määritellään oikeudet, jotka käyttäjällä on kyseiseen palveluun. (Martinelli, Nash & Topol 2015.) Esimerkkinä voisi toimia vaikkapa verkossa oleva keskustelupalsta, joka vaatii kirjautumista ennen kuin voi osallistua keskusteluun. Kirjautumisen yhteydessä käyttäjä tunnistetaan hänen syöttämänsä käyttäjätunnuksen ja salasanan perusteella ja tämän jälkeen hänet voidaan valtuuttaa

kirjoittamaan keskustelupalstalle. Toisaalta jos hänet tunnistetaan keskustelupalstan moderaattoriksi, voidaan hänelle antaa myös oikeus poistaa muiden kirjoittamia viestejä.

Kertakirjautuminen (single sign-on, SSO) on teknologia, joka mahdollistaa kirjautumisen samojen tunnusten avulla useisiin palveluihin. Näin voidaan vähentää tarvetta useille eri tunnus ja salasana yhdistelmille (Sun ym. 2013, 1). Sun ym. esittävät, että kertakirjautumisarkkitehtuuri mahdollistaa käyttäjän todentamisen ja valtuutuksen eriyttämisen, jolloin Facebookin tai Googlen kaltaiset suuret ja tunnetut toimijat voivat varmentaa käyttäjän identiteetin ja näin tarjota todentamispalveluita. Kun käyttäjä sitten on todennettu, voi palvelu itse vastata käyttäjän valtuutuksesta sen perusteella, keneksi käyttäjä vahvistettiin (Sun ym. 2013, 1). Erityisen huomionarvoista kertakirjautumisessa on, että käyttäjältä tulisi vaatia todentamista vain kerran ja tämän jälkeen kaikki palvelut tulisi olla saatavilla ilman uutta kirjautumistietojen syöttämistä (Wilson & Hingnikar 2013).

3.2 OAuth 2.0 kehys

OAuth on lyhenne sanoista Open Authorization ja se tarjoaa kehyksen, jonka avulla on mahdollista toteuttaa kertakirjautuminen eri resursseihin. Tässä keskitymme kehyksen versioon 2.0, joka tarjoaa monia parannuksia verrattuna edelliseen versioon 1.0. RFC 6749 määrittelee OAuth 2.0 kehyksen ja nimeää eri osapuolille roolit, joissa nämä toimivat. Ennen kuin ryhdymme käsittelemään OAuth 2.0 kehyksen tarjoamia mahdollisuuksia ja toimintaperiaatteita, onkin syytä käydä läpi nämä roolit kyseisestä dokumentista (Taulukko 2).

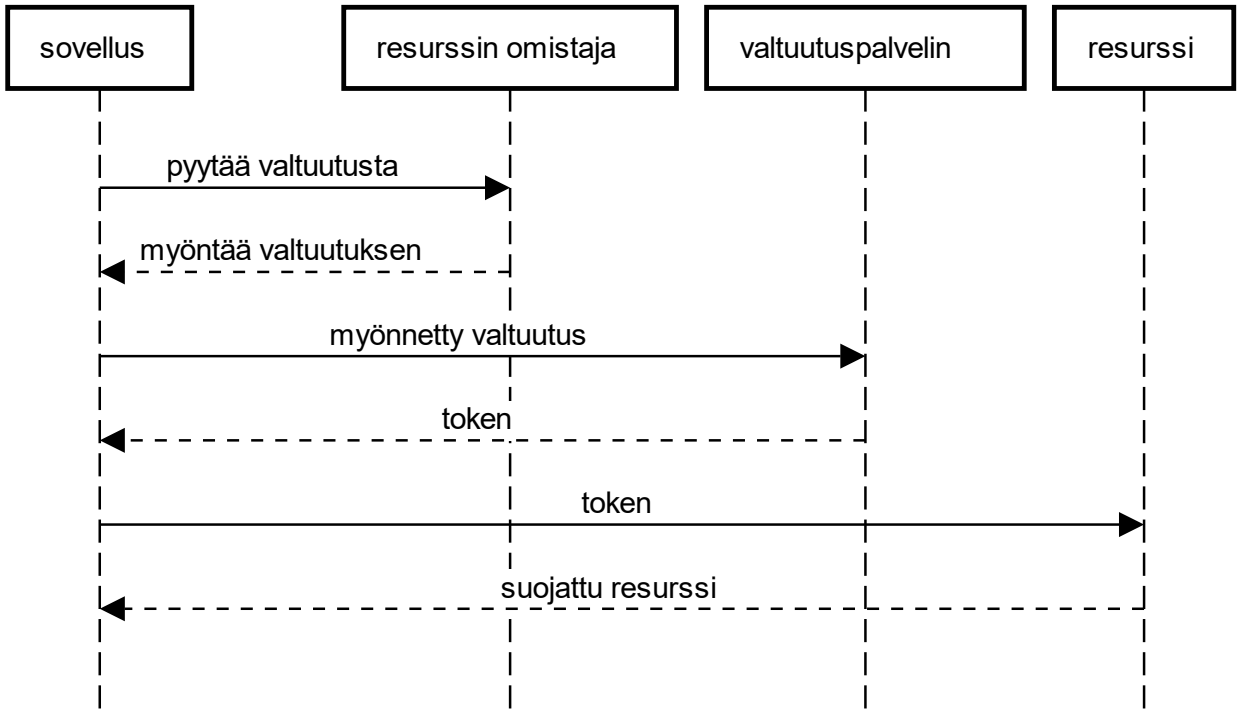
Taulukko 2. OAuth kehyksen roolit (Internet Engineering Task Force 2012, 5)

Rooli	Selite
Resurssin omistaja	Taho, jolla on valta päättää suojatun resurssin käytöstä. Yleensä käyttäjä.
Resurssipalvelin	Palvelin, jolla sijaitsee jokin resurssin omistajalle kuuluva resurssi.
Asiakas(sovellus)	Mikä tahansa sovellus, palvelu tai muu kolmas osapuoli, joka käyttää resurssipalvelimella sijaitsevaa resurssia resurssin omistajan luvalla.
Valtuutuspalvelin	Palvelin, joka todentaa resurssin omistajan ja tämän jälkeen valtuuttaa asiakkaan käyttämään resurssipalvelimella olevaa resurssia omistajan puolesta.

RFC6749 määrittelee OAuth 2.0 kehyksen periaatteet. Johdannossaan se toteaa tarjoavansa parannuksia asiakkaan ja resurssin väliseen todentamiseen, joka perinteisesti on tapahtunut siten, että resurssin omistaja on antanut salasanansa kolmannen osapuolen (=asiakas) käytettäväksi. Tässä kuitenkin piilee ilmeinen ongelma, joka todetaan myös johdannossa: kyseinen toimintamalli edellyttää, että mainittu kolmas osapuoli, eli asiakas, tallentaa saamansa salasanan luettavassa muodossa itselleen. (Internet Engineering Task Force 2012, 3.) Tästä seuraa useita merkittäviä ongelmia, jotka voidaan johtaa seurauksiksi siitä, että asiakas kolmantena osapuolena on saanut haltuunsa resurssin omistajan salasanan.

1. Koska asiakas(sovellus) käyttää nyt resurssin omistajan salasanaa ja tunnistautuu resurssille resurssin omistajana, saa tämä täydet oikeudet käyttämäänsä resurssiin. Näin ollen resurssinomistaja ei voi mitenkään rajoittaa ja tarkentaa niitä oikeuksia, jotka haluaa asiakkaalle myöntää omistamaansa resurssiin. (Internet Engineerin Task Force 2012, 3.)
2. Koska asiakas(sovellus) tietää resurssin omistajan salasanan, ei tämä voi poistaa asiakkaalta oikeutta käyttää resurssia. Ainoa tapa varmasti estää resurssin käyttäminen, on vaihtaa salasana. (Internet Engineering Task Force 2012, 3.) Tällöin kuitenkin myös kaikkien muiden asiakkaiden valtuutus päättyy, koska salasana vaihtui. Seurauksena käyttäjän on annettava uusi salasana kaikille niille sovelluksille ja palveluille, joiden haluaa yhä sallia pääsevän omistamaansa resurssiin.
3. Jos jokin valtuutetuista asiakas(sovelluksista) joutuu tietomurron kohteeksi ja sieltä varastetaan selkokielisinä tallennettu salasana, saa hyökkääjä vapaan pääsyn salasanalla suojattuun resurssiin. (Internet Engineering Task Force 2012, 4.)

Wilson ja Hingnikar (2019) toteavat, että OAuth 2.0 kehyksen ensisijainen tarkoitus on tarjota parempi vaihtoehto antaa pääsy suojattuun resurssiin, kuin luovuttaa salasana kolmannen osapuolen käyttöön. Vaikka Wilson ja Hingnikar kuvaavat aiemmin esitetyt roolit RFC 6749 mukaisesti, he käyttävät OAuth 2.0 kehyksen perusperiaatetta kuvatessaan käsitteitä hieman eri tavalla kuin RFC 6749 ne määrittelee. Wilson ja Hingnikar käyttävät asiakkaasta nimitystä sovellus ja katsovat resurssin omistajan ja sovelluksen kommunikoivan keskenään ja viittaavat resurssipalvelimeen kolmantena osapuolena (Wilson ja Hingnikar 2019). Tässä työssä sovelletaan käsitteitä siten, että jatkossa puhutaan resurssin omistajasta käyttäjänä, asiakkaasta sovelluksena tai asiakassovelluksena ja resurssipalvelimesta lyhyesti resurssina. Lisäksi katsotaan, että käyttäjä ja resurssi muodostavat parin ja sovellus on se kolmas osapuoli, jonka käyttäjä valtuuttaa käyttämään resurssia.



Kuva 1. OAuth 2.0 flow perusperiaate

Wilson ja Hignikar (2019) korostavat, että OAuth 2.0 on ennen kaikkea valtuuttamiseen tarkoitettu kehys (framework). Sen tarkoitus on ratkaista ennen kaikkea aiemmin kuvatut pulmat, jotka olivat seurausta salasanan luovuttamisesta kolmannelle osapuolelle. Tämän se tekee tarjoamalla käyttäjälle mahdollisuuden valtuuttaa kolmannen osapuolen sovelluksia käyttämään käyttäjän omistamaa resurssia. (Wilson ja Hingnikar 2019.) Kuvassa 1. kuvataan karkealla tasolla, mikä kehyksen perusajatus on. Siinä sovellus pyytää valtuutusta käyttäjältä, minkä jälkeen sovellus voi pyytää saamallaan valtuutuksella tokenin, jonka avulla se pääsee käsiksi suojattuun resurssiin käyttäjän puolesta.

RFC 6749 (2012, 8-9) määrittelee neljä tapaa antaa valtuutus sovellukselle ja jokaisella näistä tavoista on omat erityispiirteensä. Kaikissa kuitenkin keskeistä on se, että sovellukselle toimitetaan token, jonka avulla se pääsee resurssiin käsiksi. Tämä mahdollistaa periaatteessa erilaisten tokenien myöntämisen siten, että käyttäjä voi halutessaan määritellä hyvinkin tarkasti millaisen valtuutuksen sovellus tokenilla saa. Lisäksi token voidaan peruuttaa siten, että peruutus vaikuttaa vain yhdelle sovellukselle luovutettuun tokeniin ja näin kaikki muut sovellukset toimivat yhä normaalisti.

3.2.1 Asiakassovelluksen salasana

Sovelluksen salasanalla valtuutuksen hakemisessa on kyse siitä, että sovellus kirjautuu omalla sovellussalasanallaan valtuutuspalvelimelle ja hakee tokenin. Tällä tokenilla se pystyy käyttämään

suojattua resurssia. (Wilson & Hignikar 2019.) Huomionarvoista on, ettei käyttäjän tarvitse olla tekemisissä valtuutuspalvelimen kanssa (Wilson & Hignikar 2019) ja tästä syystä kyseistä valtuutustapaa käytetäänkin useimmiten silloin, kun asiakassovellus asioi omasta puolestaan (Internet Engineering Task Force 2012, 9). Karkeasti ajateltuna kuvasta 1 jää pois ensimmäinen vuorovaikutus sovelluksen ja resurssin omistajan väliltä, koska resurssi kuuluu sovellukselle itselleen.

Tämä valtuutusmenetelmä ei ole oleellinen tässä työssä käsiteltävän tuotteen näkökulmasta, joten sen käsittely laajemmin ei ole perusteltua. On kuitenkin hyvä olla tietoinen tällaisesta mahdollisuudesta, mikäli se esimerkiksi myöhemmin tulisi ajankohtaiseksi. Nyt kuitenkin keskitymme tapoihin, joilla nimenomaan tunnistetaan käyttäjä ja tarjotaan käyttäjälle mahdollisuus valtuuttaa sovellus toimimaan puolestaan.

3.2.2 Resurssin omistajan tunnistetiedot

Tässä valtuutustavassa käyttäjä syöttää omat tunnistetietonsa, eli käytännössä tunnuksen ja salasanan, sovellukseen, joka lähettää ne eteenpäin valtuutuspalvelimelle hakeakseen tokenin suojatun resurssin käyttöön. Toteutustapa, jossa tunnuksilla haetaan token, on askel parempaan tilanteesta, jossa sovellus vain tallentaisi tunnistetiedot itselleen jatkokäyttöä varten. (Internet Engineering Task Force 2012, 9.) Haasteena tosin on, että valtuutuspalvelin ei pelkän käyttäjätunnuksen ja salasanan perusteella pysty sanomaan, onko kirjautuja käyttäjä itse vai hänen puolestaan toimiva sovellus. Jos siis halutaan rajoittaa sovelluksen oikeuksia, täytyy sovelluksen itsensä ottaa tästä vastuuta (Wilson & Hignikar 2019).

Wilson ja Hignikar (2019) mukaan valtuutustapaa ei suositella, koska käyttäjä paljastaa siinä henkilökohtaiset tunnistetietonsa kolmannelle osapuolelle, eli valtuutusta vaativalle sovellukselle. RFC 6749 toteaakin, että toteutuksen edellytys on vahva luottamus resurssin omistajan ja sovelluksen välillä. RFC 6749 mukaan resurssin omistajan tunnistetietoihin perustuvaa valtuutusta tulisikin käyttää vain silloin, kun muita ratkaisuja ei ole mahdollista käyttää. (Internet Engineering Task Force 2012, 9.)

Ratkaisu on kuitenkin hyvin helppo toteuttaa, koska siinä tarvitaan käytännössä vain asiakassivustoon tai -sovellukseen upotettu kirjautumiskenttä, jonne käyttäjä syöttää tunnuksensa ja salasanansa. Saatuaan nämä tiedot, sovellus pystyy kirjautumaan niillä suoraan käyttäjän puolesta. Kokemus on käyttäjälle helppo, eikä vaadi teknisestikään kovin monimutkaista toteutusta. Kuva 1 kuvaa todella hyvin tätä ratkaisua. Käytännössä resurssin omistajan myöntämänä valtuutuksena toimivat hänen henkilökohtainen käyttäjätunnuksensa ja salasanansa.

3.2.3 Implisiittinen valtuutus

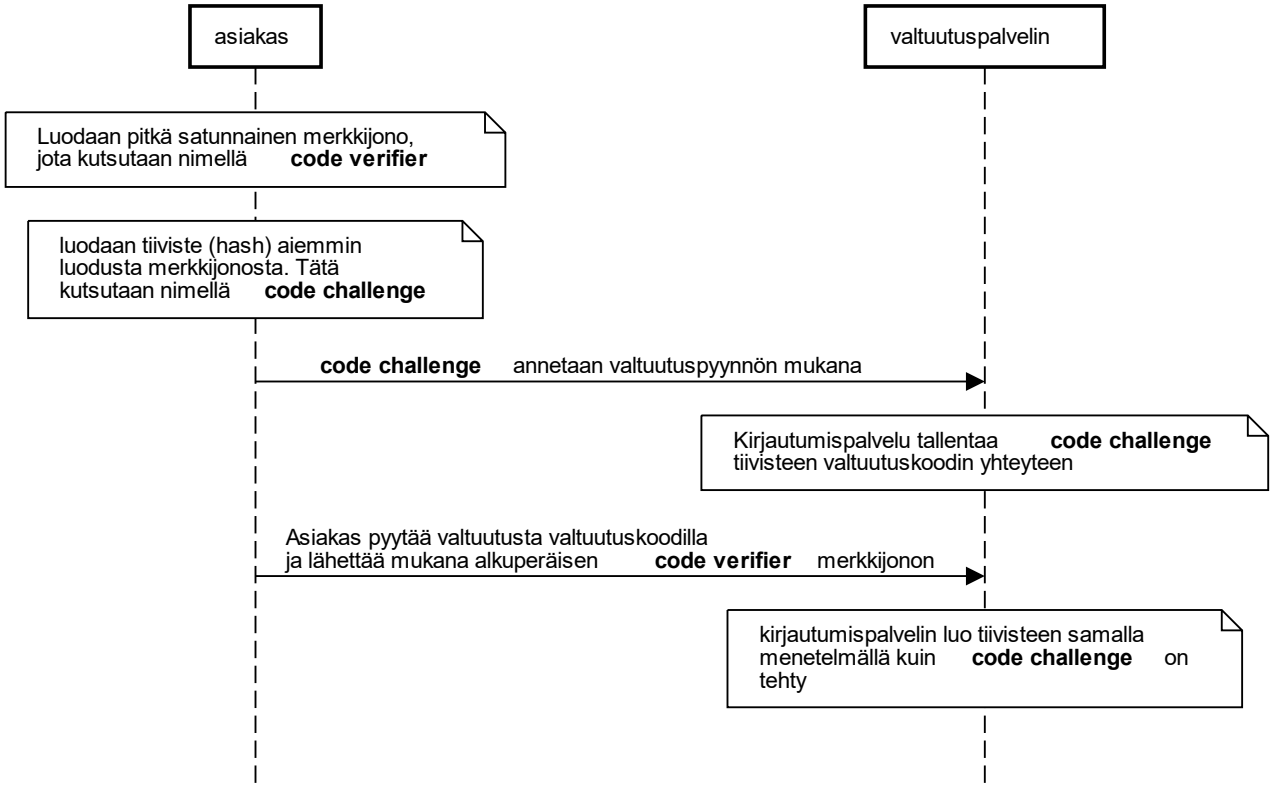
RFC 6749 kuvailee implisiittisen valtuutuksen siten, että käyttäjä ohjataan valtuutusta tarvitsevasta asiakassovelluksesta valtuutuspalvelimelle, joka tunnistaa käyttäjän ja allekirjoittaa tälle tokenin. Käyttäjä palaa tämän jälkeen asiakassovellukseen mukanaan token, joka valtuuttaa sovelluksen suojattuun resurssiin. (Internet Engineering Task Force 2012, 8-9.) Tämä tuo merkittäviä parannuksia siihen, että käyttäjä luovuttaisi kirjautumistietonsa suoraan asiakassovellukselle. Wilson ja Hignikar (2019) tarkentavat, että tämä valtuutustapa olisi kehitetty ratkaisuksi kiertää CORS-ongelmat aikana, jolloin kaikki selaimet eivät vielä tukeneet CORSia. Tähän näkemykseen voi kuitenkin suhtautua kriittisesti siinä mielessä, että CORS mahdollistaisi implisiittisen kirjautumisen korvaamisen sillä, että käyttäjä syöttäisi tunnistetietonsa suoraan asiakassovellukseen. Tällöin päädytään aiemmin kuvattuun resurssin omistajan tunnistetietoja käyttävään valtuutusratkaisuun ja törmätään kaikkiin sen ongelmiin.

Joka tapauksessa implisiittinen valtuutustapa on toimiva etenkin SPA-sovelluksissa, joilla ei ole omaa palvelinpuolen ohjelmistoa. Tällöin käyttäjä voi hakea tokenin suoraan valtuutuspalvelimelta ja luovuttaa sen SPA-sovellukselle. On kuitenkin tärkeää huomata, että siirtymät tapahtuvat uudelleenohjauksilla ja tietoa välitetään osoiterivillä. Tällöin on vaarana, että token jää esimerkiksi lokitietoihin tai sivuhistoriaan, mistä se voi päätyä vääriin käsiin. (Wilson & Hignikar 2019.)

3.2.4 Valtuutuskoodi

Valtuutuskoodiin perustuva valtuuttaminen on jokseenkin samankaltainen kuin implisiittinen valtuutustapa, mutta se tarjoaa parannuksia tietoturvaa ja lisää aavistuksen toteutuksen monimutkaisuutta. Se perustuu hyvin vahvasti selaimen uudelleenohjaukseen, mutta sen sijaan, että käyttäjä tunnistautumisen jälkeen palaisi valtuutuspalvelimelta tokenin kanssa, palaakin käyttäjä valtuutuskoodin kanssa. Saatuaan valtuutuskoodin, sovellus voi hakea tokenin suoraan valtuutuspalvelimelta käyttäen kyseistä valtuutuskoodia. Näin token ei kulje käyttäjän kautta, eikä sitä näin ollen voi kukaan varastaa matkalla. (Internet Engineering Task Force 2012, 8; Wilson & Hignikar 2019.)

Koska valtuutuskoodi kuitenkin kulkee osoiterivillä ja voi päätyä vääriin käsiin, täytyy jotenkin varmistaa ettei koodiin käsiksi päässyt ulkopuolinen taho voi noutaa tokenia koodia käyttäen. Tätä varten käytetään PKCE mekanismia. PKCE tulee sanoista proof key for code exchange, eikä RFC 6749 tunne sitä lainkaan. Sen sijaan PKCE määritellään omassa dokumentissaan RFC 7636. (Internet Engineering Task Force 2012; Internet Engineering Task Force 2015b.)



Kuva 2. PKCE flow

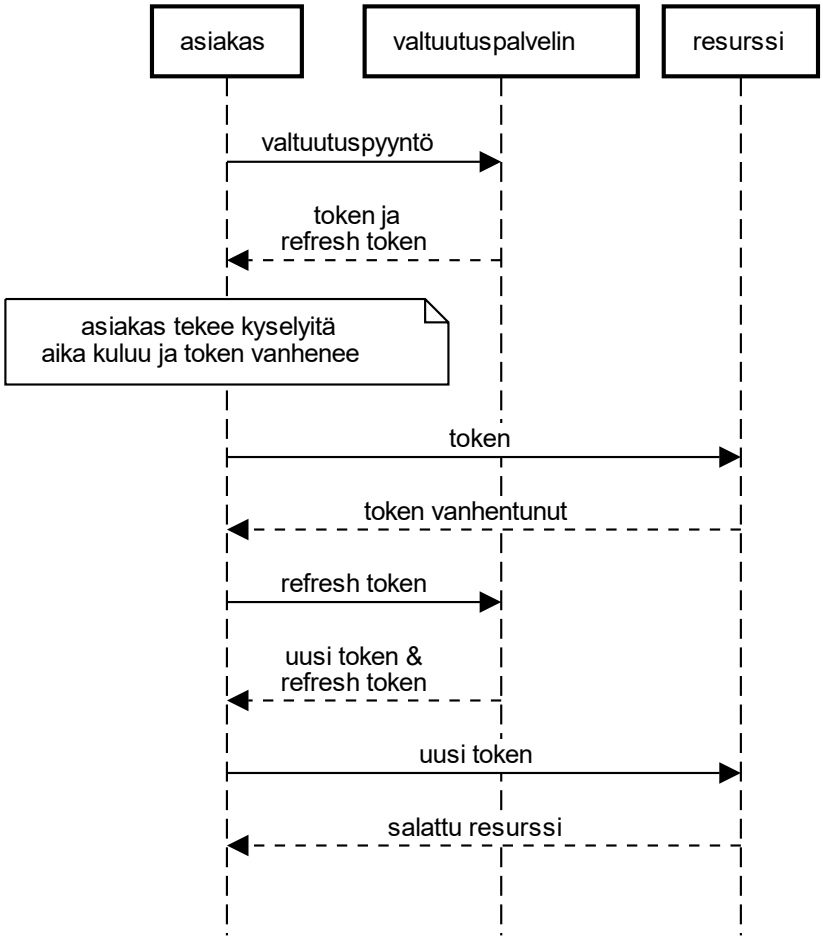
Wilson ja Hignikar kuvaavat PKCE-menetelmää kuvassa 2. näkyvällä tavalla. Siinä asiakas luo ensin satunnaisen merkkijonon ja tämän jälkeen RFC:ssä 7636 määritellyllä tavalla luo siitä tiivisteen (hash). Tämän tiivisteen käyttäjä toimittaa valtuutuspalvelimelle tullessaan tunnistautumaan. Kun tunnistautuminen on onnistunut, valtuutuspalvelin luo valtuutuskoodin, jolla tokenin voi hakea. Valtuutuspalvelin tallentaa tämän luomansa valtuutuskoodin ja käyttäjän sille tuoman tuoman tiivisteen odottamaan. Kun asiakassovellus tulee hakemaan tokenia valtuutuskoodilla, tämä lähettää koodin mukana alkuperäisen generoimansa merkkijonon. Koska tiiviste on tehty yksisuuntaisella algoritmilla, voidaan olettaa, ettei kukaan muu kuin valtuutusta alun perin pyytänyt asiakassovellus tiedä merkkijonoa, josta tiiviste oli tehty. Näin valtuutuskoodi yksinään ei riitä tokenin hakemiseen, eikä siis sen varastaminen riitä valtuutuksen saamiseen. (Wilson & Hignikar 2019.)

3.2.5 JSON web token

Tässä työssä tokeneilla tarkoitetaan JSON web tokeneita (JWT), jotka määritellään RFC:ssa 7519. JWT mahdollistaa tiedon välittämisen HTTP-pyynnön otsaketiedoissa siten, että tieto on helposti parsittavassa JSON-muodossa ja välitetty tieto on mahdollista varmentaa luotettavaksi. Käytännössä tämän työn näkökulmasta olennaista on ymmärtää, että token tarjoaa mahdollisuuden välittää tietoa siitä, kuka HTTP-pyyntöjä tekee ja vieläpä siten, että esitty väite pyynnön tekijästä on

riittävän luotettava. Riittävän siinä mielessä, että oletuksena on, ettei tokenin allekirjoittamiseen käytetty avain ole päätynyt vääriin käsiin. (Internet Engineering Task Force 2015a.)

RFC 7519 määrittelee tokenin sisältöön exp-nimisen kentän, jonka käyttö tosin ei ole pakollista. Kyseisessä kentässä määritellään, milloin token vanhenee ja tämän jälkeen sen ei tulisi enää kelvata tunnistamiseen eikä siinä kuljetettua tietoa tule pitää luotettavana. Kentän voi jättää pois, jolloin token on käytännössä ikuinen. (Internet Engineering Task Force 2015a.) Tässä työssä tokenit toimivat kuin avaimet: ne sallivat pääsyn tiettyyn resurssiin. Näin ollen on tärkeää, että avaimia ei vain jatkuvasti luoda uusia vaan varmistetaan myös vanhojen poistuminen kierrosta. Tästä syystä tokeneissa käytetään exp-kenttää, jonka avulla vanhentuneet tokenit voidaan hylätä.



Kuva 3. Refresh tokenin käyttö

Tokenien vanheneminen aiheuttaa kuitenkin ongelman: mitä jos token vanhenee, vaikka sillä halutaisiin tehdä vielä ihan luvallisia pyyntöjä? Tätä varten RFC 6749 määrittelee OAuth 2.0 kehykseen refresh tokenin, jonka avulla voidaan hakea uusi token edellisen mentyä vanhaksi. Tämä toimenpide voidaan tehdä taustalla, jolloin käyttäjälle ei aiheudu vaivaa. Se kuitenkin vaatii tuen OAuth toteutukselta ja tällöin tulee varsinaisen tokenin yhteydessä antaa myös mainittu refresh token.

Refresh token täytyy lisäksi hyväksyä, mikäli valtuutuspalvelimelle tällainen toimitetaan ja sitä vastaan tulee antaa uusi token-pari. Tapahtuma kuvataan karkealla tasolla kuvassa 3. (Internet Engineering Task Force 2012, 10-11.)

4 Toteutus

Toteutuksen kuvailussa käytetään runsaasti aikaa suunnittelun kuvailemiseen, koska tämän havaittiin olevan keskeinen vaihe sovelluksen onnistuneessa toteutuksessa. Sen lisäksi kuvaillaan kehitysprosessin vaiheita ja sitä, miten vaatimukset ja tavoitteet muuttuivat prosessin aikana. Esiin nostetaan myös toteutetun ratkaisun yksilölliset ominaispiirteet ja käydään läpi lopullisen tuotoksen rakennetta. Luvun keskeiset käsitteet on kuvattu taulukossa 3.

Taulukko 3. Luvun 4 keskeiset käsitteet

Käsite	Selitys
API	API on lyhenne sanoista Application Programming Interface ja se on rajapinta, jonka kautta sovelluksen kanssa voidaan kommunikoida. Tässä yhteydessä sillä tarkoitetaan luvussa 2.1 kuvailtua mikropalvelun julkista rajapintaa.
Docker	Docker on virtualisointiohjelmisto, jonka avulla voidaan suorittaa sovelluksia eristetyssä ympäristössä.
Integraatiotesti	Integraatiotestillä viitataan tässä yhteydessä testiin, joka testaa useiden yksittäisten toiminnallisuuksien yhteistoimintaa.
Man in the middle -hyökkäys	Tällä yhteydessä tällä hyökkäyksellä tarkoitetaan lähinnä tilannetta, jossa ulkopuolinen taho salakuuntelee kahden muun osapuolen kahdenkeskistä verkkoliikennettä.
Postgresql	Postgresql on avoimeen lähdekoodiin tietokantajärjestelmä.
Yksikkötesti	Yksikkötestillä testataan yksittäisen koodiyksikön oikeanlaista toimintaa.

4.1 Suunnittelu

Tässä luvussa määritellään tuotoksen onnistumisen mittarit ja tehdään vaatimusmäärityst. Vaatimusmääritysten muodostamiseen on käytetty käyttäjätarinoita, joista voidaan johtaa tarvittavat toiminnallisuudet ohjelmointityön ohjaamiseen. Myös kehitysympäristön pystyttämistä kuvaillaan

lyhyesti ja tässä yhteydessä tuodaan myös esiin sitä, miten sovelluksen testiautomaatio on pyritty huomioimaan jo kehityksen alkuvaiheessa.

4.1.1 Lähtötilanne

Tuotetta lähdettiin kehittämään yksityishenkilölle, joka on harrastanut ohjelmointia ja etenkin web-kehitystä. Monet hänen projektinsa tarvitsevat jonkinlaisen kirjautumistoiminnon. Tämä tarkoittaa sitä, että jokaisessa projektissa joudutaan käyttämään aikaa siihen, että kirjoitetaan uudelleen käyttäjänhallinta ja valitaan tarvittava tietokanta ja mietitään, miten salasanat suojataan. Erityisen huomionarvoista on, että monet näistä projekteista syntyivät omista käyttötarpeista, jolloin jokaiseen palveluun täytyi tehdä erilliset tunnukset. Tästä seuraa valtava määrä eri tunnuksia ja vaikka jokaiseen palveluun laittaisi saman käyttäjätunnuksen ja salasanan, vaatii salasanan vaihto toimenpiteitä jokaisessa palvelussa erikseen.

Tuotoksen on siis tarkoitus ratkaista ongelma lukuisista käyttäjätunnuksista tarjoamalla keskitetyn ratkaisun. Aiemmin erilaisten sovellusten ja palveluiden kehittämisessä pääpaino on ollut uudessa tuotteessa ja pakollinen käyttäjähallinta on tehty vain nopeasti tämän tuotoksen kylkeen. Nyt käyttäjähallinta päätettiin ottaa keskiöön ja se haluttiin toteuttaa kerran huolellisesti, jotta jatkossa muut palvelut voivat nojata toteutettavaan keskitettyyn ratkaisuun. Kaikki käyttäjätunnukset ja salasanat keskitetään uuteen palveluun ja siellä huolehditaan niiden asianmukaisesta salaamisesta ja varastoinnista, jolloin toivotaan myös tietoturvan paranevan.

Tuotoksen katsotaan olevan onnistunut, kun käyttäjä voi yhdellä käyttäjätunnuksella kirjautua lukuisiin eri palveluihin helposti ja vaivattomasti. Lisäksi käyttäjän täytyy pystyä vaihtamaan salasanansa siten, että hän voi välittömästi salasanan vaihdettuaan kirjautua kaikkiin palveluihin uudella salasanallaan. Tietoturvan näkökulmasta tärkeä kriteeri on, ettei salasana pääse vuotamaan muihin palveluihin keskitetystä käyttäjätietokannasta. Lisäksi käyttäjä tulee suojata siten, ettei ulkopuolinen taho pysty aiheuttamaan vahinkoa käyttäjän tunnukselle, vaikka saisi tämän houkuteltua pahantahtoiselle sivustolle tai klikkaamaan väärennettyä linkkiä. Viimeisenä tärkeänä vaatimuksena on käyttäjän tietosuoja, joka halutaan nostaa keskiöön keskitetyssä ratkaisussa. Käyttäjällä tulee olla kontrolli siitä, mitä tietoja hänestä luovutetaan ja mihin. Lisäksi käyttäjällä tulee olla helppo keino poistaa omat tietonsa kaikkialta niin halutessaan.

4.1.2 Vaatimusmääritys

Suunnittelu aloitettiin pohtimalla, missä tilanteissa kyseistä palvelua tarvitaan ja millaisia ominaisuuksia siltä vaaditaan. Käytännössä tämä tapahtui kirjoittamalla käyttäjätarinoita, joiden pohjalta on mahdollista muodostaa kuva siitä, millaisia tarpeita tulevalle palvelulle on ja miksi tällaisia ominaisuuksia tarvitaan. Käyttäjätarinoita laatiessa on olennaista hahmottaa ne tahot, jotka palvelua

käyttävät. Vasta tämän jälkeen on mahdollista pohtia, mitä nämä haluavat tehdä ja miksi. Hyvä käyttäjätarina vastaakin kysymyksiin kuka, mitä ja miksi (Fowler 2013). Fowlerin (2013) mukaan hyvä käyttäjätarinan rakenne on "As a ... I want ... So that ...", jonka voisi vapaasti suomentaa ”[käyttäjä]nä/-na… haluan… jotta…”. Yksi esimerkki näin syntyneistä käyttäjätarinoista on seuraava:

> "Käyttäjänä haluan käyttää samaa salasanaa kaikissa palveluissa, jotta minun ei tarvitse muistaa kuin yksi salasana. "

On tärkeää huomioida, että palvelua voivat käyttää ihmisten lisäksi myös toiset palvelut. Tällöin täytyy miettiä, millaisiin palveluihin kyseinen palvelu integroituu ja mitä nämä palvelut tarvitsevat uudelta palvelulta. Myös näihin tilanteisiin voi hyvin kirjoittaa käyttäjätarinoita, mutta tällöin käyttäjänä toimii toinen palvelu. Esimerkkinä tällaisesta tarinasta on seuraava:

> "Palveluna en halua ylläpitää käyttäjätietokantaa, jotta minun ei tarvitse huolehtia salasanojen turvallisesta säilyttämisestä. ”

Käyttäjätarinoiden kirjaaminen on koettu yhdeksi keskeisistä vaiheista kehitystyössä, koska niiden huolellinen suunnittelu etukäteen vähentää riskiä uusien vaatimusten paljastumiselle kesken kehitystyön. Tässä projektissa syyllistyttiin tekemään hieman liian vähän käyttäjätarinoita, jolloin vaatimusmäärityksistä tuli melko kapeat. Niiden pohjalta saatiin kuitenkin valittua tietokannaksi Postgresql tietokanta, jolle laadittiin relaatiomalli ja kirjattiin ylös taulut, jotka tietokantaan täytyy tehdä.

Tässä kohdassa käytettäväksi ohjelmointikieleksi valittiin Typescript ja valittiin ne keskeiset tarvittavat kirjastot, joista tämä tuote tulisi olemaan riippuvainen. Merkittäviä ratkaisuja olivat päätös käyttää Node.js:n natiivia HTTP-moduulia suosittujen web-kehysten sijaan ja näin vähentää riippuvuuksia. Lisäksi tietokantaa päätettiin käyttää kirjoittaen sql-kyselyt itse sen sijaan, että käytettäisiin jotain kirjastoa abstrahoimaan tietokantakyselyt. Tästä syystä myös migraatioita varten jouduttiin kirjoittamaan tarvittava koodi, joka huolehtii tietokantataulujen ajantasaisuudesta ja tarvittaessa aiempaan tilaan palauttamisesta. Toteutus on yksinkertainen, mutta riittää hyvin tämän pienen palvelun tarpeisiin.

Vaikka valmiiden kirjastojen käyttö pyrittiin minimoimaan, joihinkin asioihin oli mielekästä kuitenkin valita valmis tuote. Etenkin kaikki salaamiseen liittyvä koodi otettiin valmiina kirjastoina tietoturvasyistä. Näiden lisäksi datan validointiin valittiin valmis kirjasto. Datan validoiminen on tärkeä vaihe, koska avoimeen internetiin kytkettyyn palvelimeen voi kuka tahansa lähettää minkälaisia syötteitä tahansa. Siksi palvelimen tulee tarkistaa perusteellisesti kaikki saapuva data ja varmistaa, että se on oikeassa muodossa. Tämä katsottiin niin kriittiseksi osaksi, että myös siinä päädyttiin käyttämään valmista kirjastoa virheiden välttämiseksi.

4.1.3 Kehitysympäristön pystyttäminen

Heti alussa luotiin projektille oma Github-repositorio, jonne kirjattiin alusta asti kaikki suunnittelussa syntyvä dokumentaatio. Tämä siitä syystä, että Github tarjoaa automaattisesti versioinnin ja toisaalta kaiken vieminen repositorioon takaa, että tieto pysyy kootusti yhdessä paikassa. Samalla muodostuu automaattisesti pohjadokumentaatio, jonka päälle dokumentaatiota on hyvä laajentaa sitä mukaa, kun projekti etenee. Näin pienessä projektissa ei koettu mielekkääksi hajauttaa dokumentaatiota ja vaatimusmäärityksiä toiselle alustalle vaan pitää ne koodin lähellä.

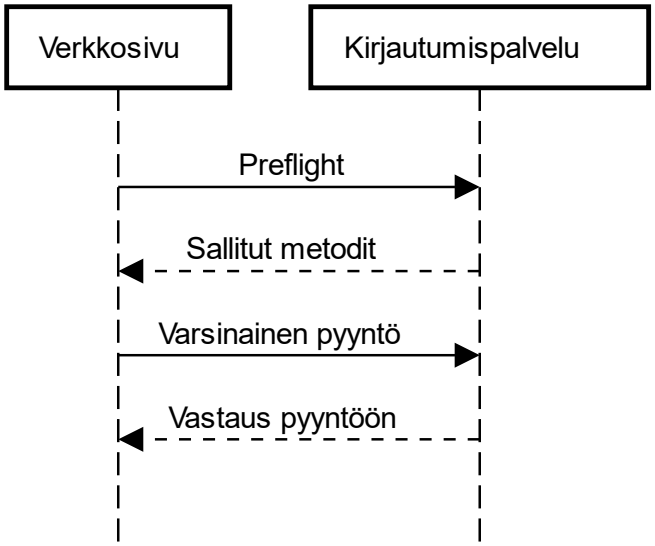
Ensimmäinen vaihe projektin aloittamisessa oli luonnollisesti hakemistorakenteen luominen ja tuotantoautomaation toteuttaminen. Tämä tarkoittaa testausautomatiikan pystyttämistä, johon myöhemmin voidaan lisätä myös julkaisuautomaatio. Tässä projektissa päätettiin toteuttaa hakemistorakenne siten, että se jakautuu kahteen päähakemistoon: src ja test. Varsinainen ohjelmistokoodi on hakemistossa src ja kaikki testit sijoitetaan test hakemiston alle joko alihakemistoon Integration tai Unit. Yksikkötestit sijoitetaan Unit hakemistoon, jonka sisältö noudattaa samaa rakennetta kuin hakemiston src sisältö. Tällöin jokaiselle kooditiedostolle on helppo löytää sitä vastaavat yksikkötestit, mutta tarvittaessa koodi voidaan kopioida eteenpäin myös ilman testejä. Integraatiotestien osalta samanlaista hakemistorakennetta ei ollut mielekästä toteuttaa, koska niiden on tarkoitus testata kokonaisuuksia. Integraatiotestit jakautuivat kuitenkin kahden tyyppisiin testeihin: toisissa testattiin kokonaisia tapahtumaketjuja käynnistämällä koko sovellus ja toisissa testattiin yksittäisten pyyntöjen käsittelyä rajapinnassa, mutta kuitenkin aina tietokantaan asti.

Koska tietokantana oli Postgresql eikä sitä ollut piilotettu minkäänlaisen abstrahoivan kirjaston alle, täytyi kehittäessä pystyä tekemään kyselyitä nimenomaan Postgresql tietokantaan, jos sovellusta halusi kokeilla käytännössä tai ajaa integraatiotestejä. Tästä syystä projektin yhteyteen luotiin myös tarvittavat tiedostot, jotta kehitysympäristön sai helposti käyntiin Dockerin avulla. Näin kehittäjän ei tarvinnut käyttää aikaa siihen, että rakentaa kehitysympäristöä tarvittavine tietokantapalveluineen esimerkiksi tietokonetta vaihtaessa tai jos projektiin liittyisi uusi kehittäjä.

4.2 Ensimmäinen versio

Ensimmäisessä versiossa lähdettiin toteuttamaan hyvin yksinkertaista ratkaisua, jossa kirjautumiskenttä upotetaan kirjautumispalvelua käyttävään sovellukseen tai sivustoon. Tällöin kirjautumispalvelua käyttävän sovelluksen toiminnallisuudesta vastaisi Javascript-koodi, joka lähettäisi kirjautumistiedot kirjautumispalvelulle ja tallentaisi vastaanottamansa tokenin tulevaa käyttöä varten. Ratkaisu vastaisi siis OAuth 2.0 määritelmässä kuvailtua resurssin omistajan salasanalla kirjautumista, vaikka tässä vaiheessa toteutuksen perustana ei ollutkaan kyseinen RFC 6749.

Toteutus ehdittiin saada toimivaksi asti. Se oli suhteellisen helppo rakentaa, koska se vaati vain palvelinohjelmiston rakentamisen ja API-rajapinnan tarjoamisen eri sovelluksille, jotta nämä voivat kirjata käyttäjän sisään tätä rajapintaa vasten. Tässä ratkaisussa oli siis käytännössä yksi palvelinsovellus, joka olisi toiminut yhtenä itsenäisenä mikropalveluna palvellen lukuisia muita sovelluksia. Tietokantatoiminnallisuus oli rakennettu sisään sovellukseen ja erillistä käyttöliittymää ei tarvittu, joskin tuki staattisten sivujen tarjoamiselle päätettiin lisätä, jotta voitaisiin tarjota yksinkertainen sivu salasanan vaihtamiseen. Huomionarvoista tässä toteutuksessa oli se, että modernit selaimet estävät Javascriptillä tehtävät pyynnöt toiselle palvelimelle (MDN Web Docs 2022a). Tästä syystä jokaista uutta palvelua varten kirjautumispalveluun täytyi lisätä tieto kirjautumispalvelua käyttävästä palvelusta ja määritellä, mitä metodeja kyseisestä palvelusta sallitaan. Tämän avulla pystyttiin määrittelemään cross-origin resource sharing -sääntöjä, joiden avulla selaimet osaavat sallia luvalliset pyynnöt kirjautumispalveluun. Kuvassa 4 verkkosivu haluaa tehdä kirjautumispyynnön toisessa osoitteessa olevaan kirjautumispalveluun, jolloin selain tekee ensin preflight-pyynnön ja vasta tämän jälkeen suostuu tekemään varsinaisen pyynnön, mikäli kirjautumispalvelu ilmoittaa sen sallituksi.



Kuva 4. Pyynnön luvallisuus varmistetaan ensin preflight-pyynnöllä

Ensimmäisessä versiossa äärimmäisen tärkeäksi muodostuu CORS, jonka kanssa täytyi olla todella tarkkana. Pienikin virhe johti helposti siihen, että kirjautuminen ei toiminut tai virhetilanteita ei saatu hoidettua asiallisesti. Tämän olivat havainneet myös Larrucea, Santamaria, Colomo-Palacios & Ebert (2018, 98-99), jotka olivat huomanneet millaisia haasteita aiheutui, kun verkkosivun täytyi tehdä kyselyitä lukuisiin ulkopuolisiin palveluihin. Nämä ongelmat on kuitenkin mahdollista välttää

toteuttamalla mikropalvelut siten, että palvelimet keskustelevat keskenään sen sijaan, että verkkosivu on suoraan yhteydessä lukuisiin palveluihin. Se kuitenkin vaatii hieman toisenlaisen lähestymistavan.

CORS-haasteiden lisäksi alkoivat pian paljastua myös muut haasteet liittyen upotettuun kirjautumisikkunaan, jossa siis token haettiin käyttäen resurssin omistajan käyttäjätunnusta ja salasanaa. Ilmeistä oli, että kirjautumista vaativaan palveluun oli pakko luottaa - saihan se halutessaan käsiinsä käyttäjän salasanan luettavassa muodossa. Lisäksi toteutus ei varsinaisesti täyttänyt kertakirjautumista käsittelevässä kappaleessa todettua kertakirjautumisen määritelmää, koska tunnukset piti syöttää erikseen jokaiseen palveluun, johon haluttiin kirjautua. Viimeisimpänä nousi huoli käyttäjän tietoturvasta, sillä vaikka CORS mahdollistaa määritellä, mitkä metodit ovat sallittuja, täytyi ainakin jossakin olla palvelu, joka sallii sellaisia pyyntöjä kuten PATCH, PUT ja DELETE. Tällöin palvelu, jolle nämä sallitaan, tarjoaa potentiaalisen hyökkäysalustan erilaisille cross-site scriptin hyökkäyksille. Esitetyistä syistä päätettiin luopua jo toteutetusta upotetusta kirjautumiskentästä ja siirryttiin toteuttamaan valtuutuskoodiin perustuvaa ratkaisua valtuutuksen toteuttamiseksi.

4.3 Todellinen kertakirjautuminen

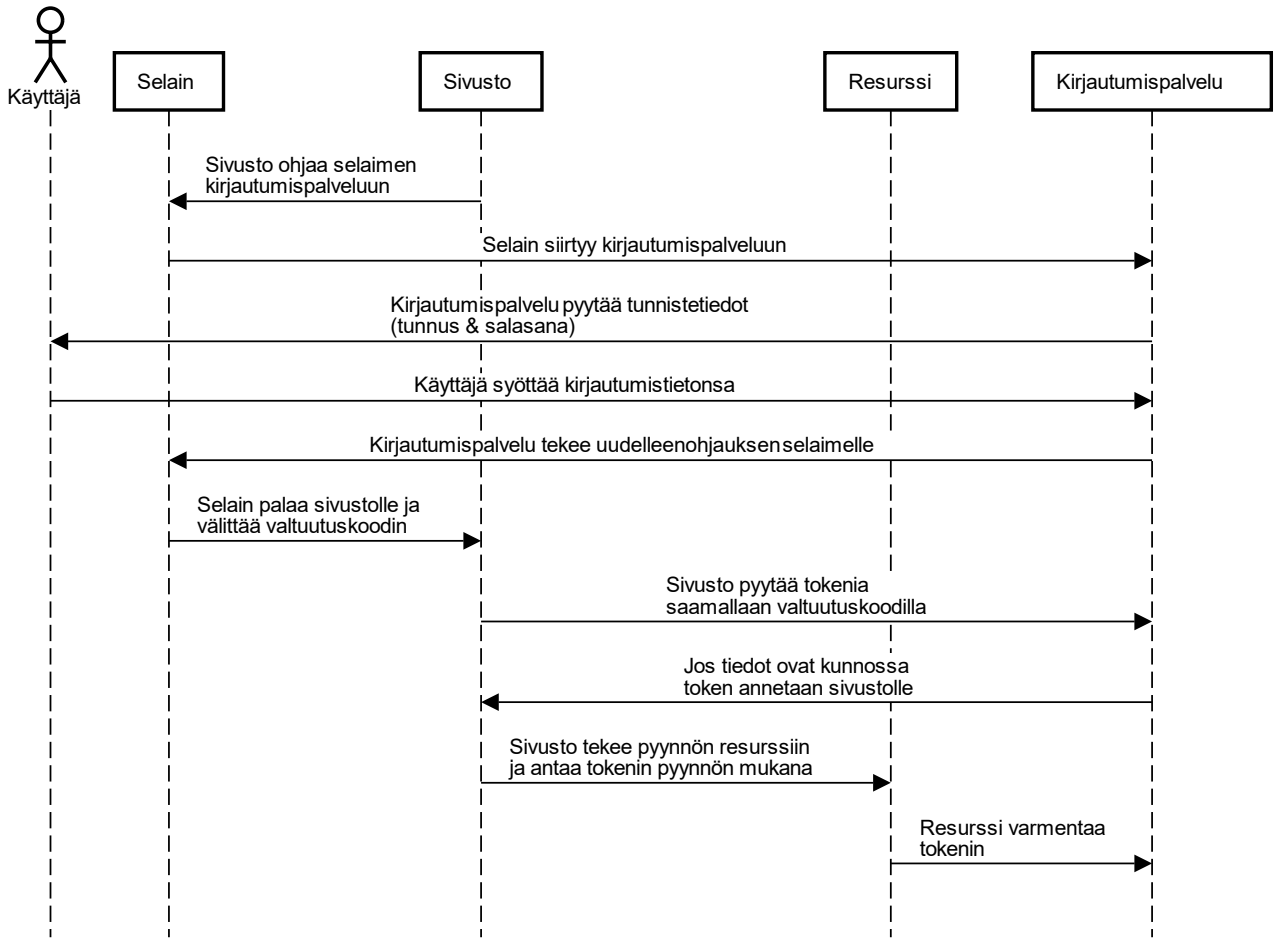
Ensimmäisen version pulmia tutkittaessa törmättiin RFC 6749:n määrittelemään OAuth 2.0 kehykseen ja määritelmän tarkempi tutkiminen johti siihen, että päätettiin toteuttaa valtuutuskoodiin perustuva ratkaisu kyseisestä OAuth 2.0 kehyksestä. RFC 6749 on perusteellinen ja se tarjoaa yksityiskohtaiset ohjeet toteutukseen. RFC:n tukena käytettiin myös kirjallisuutta, joka käsitteli perusteellisemmin flowta sekvenssikaavioiden avulla. Tässä vaiheessa jouduttiin myös tarkastelemaan uudestaan tuotteen omaa dokumentaatiota, joka ei enää täsmännyt uuden toimintatavan kanssa. Jouduttiin rakentamaan uudestaan tietokantataulut ja samalla tekemään muutoksia API-dokumentaatioon, kun uusia endpointteja lisättiin ja vanhoja poistettiin. Tämä johti kohtalaisen suureen päivitystarpeeseen dokumentaatiossa ja asetti myös käyttäjätarinat uudelleen tarkasteluun.

Uuden toimintamallin valinta johti siihen, että toteutettava ohjelmisto jouduttiin määrittelemään uudestaan alusta asti. Tämä ei kuitenkaan tarkoittanut, että kaikki vanha olisi ollut turhaa ja kaikki olisi aloitettu alusta. Koodissa oli suuria muutostarpeita, mutta suurin osa koodista oli silti siirrettävissä enemmän tai vähemmän sellaisenaan uuteen toteutustapaan. Aiemmin käyttöliittymän tehtävää oli ajanut yksi staattinen tiedosto, jonka kautta oli voinut vaihtaa salasanansa. Nyt käyttöliittymän tarpeet monipuolistuivat ja se siirrettiin omaksi tuotteekseen ja sille perustettiin oma repositorio. Näin koodi alkoi myös hajota pienemmiksi itsenäisiksi osiksi yhden monoliitin sijaan. Toteutus alkoi muuttua enemmän mikropalvelumaiseksi.

Tämä siirtyminen valtuutuskoodin käyttöön johti myös merkittäviin parannuksiin niin tietoturvan kuin käyttäjäkokemuksenkin näkökulmasta.

Tietoturvan näkökulmasta merkittävä parannus oli se, että käyttäjä ei enää syöttänyt mitään tietojaan kirjautumispalvelua käyttävään asiakassovellukseen, vaan sovelluksessa oli linkki, joka ohjasi käyttäjän kirjautumispalveluun. Linkin parametrien mukana välitettiin tieto siitä, mistä käyttäjä oli saapunut kirjautumaan ja minne hänet tuli ohjata takaisin kirjautumisen jälkeen. Lisäksi välitettiin dynaamisesti muodostettuja arvoja, joiden avulla suojattiin sekä käyttäjää, että kirjautumispalvelua erilaisilta hyökkäyksiltä, kuten cross site request forgery ja man in the middle -hyökkäyksiltä. Valtuutuskoodiin perustuvassa kirjautumisessa huomionarvoista on myös se, ettei tokenia kuljeteta sitä tarvitsevalle sovellukselle käyttäjän selaimen kautta, jolloin voidaan paremmin varmentaa, ettei se päädy sivullisille tahoille.

Myös käyttäjäkokemus paranee, koska kirjautumistietoja ei tarvitse syöttää toistuvasti uudestaan. Kirjautumispalvelu voidaan toteuttaa siten, että se tallentaa oman sessionsa käyttäjän selaimeen. Tässä tuotteessa päädyttiin käyttämään toteutukseen local storagea (MDN Web Docs 2022b), jonne tallennettu tieto säilyy, vaikka selain suljettaisiin tai sivustolta poistuttaisiin välissä. Tällöin käyttäjä voidaan tunnistaa näiden tietojen pohjalta ja riittää, kun luodaan valtuutuskoodi ja ohjataan käyttäjä takaisin valtuutusta pyytäneeseen sovellukseen sen kanssa.



Kuva 5. Tunnistautuminen ja valtuutus valtuutuskoodin kanssa

Kuvassa 5 näkyy karkea yksinkertaistus siitä, miten valtuutus myönnetään valtuutuskoodiin pohjautuvassa mallissa. Periaatteessa sivuston sijaan kyseessä voisi olla mikä tahansa sovellus, mutta tässä vaiheessa palvelu palvelee vain erilaisia verkkosivustoja selaimen välityksellä. Kuten kuvasta näkyy, käyttäjän henkilökohtaiset tunnukset eivät ikinä päädy sivuston saataville, vaan ne annetaan suoraan kirjautumispalvelulle. Huomionavoista on myös, että kirjautumispalvelu varmistaa käyttäjältä, mitä tietoja käyttäjästä annetaan valtuutusta pyytäneelle sivustolle ja millaiset oikeudet sivustolle myönnetään sen pyytämään resurssiin. Näin käyttäjä tietää tarkasti, mitä tietoja hänestä välitetään eteenpäin ja mitä oikeuksia hän on myöntämässä sivustolle.

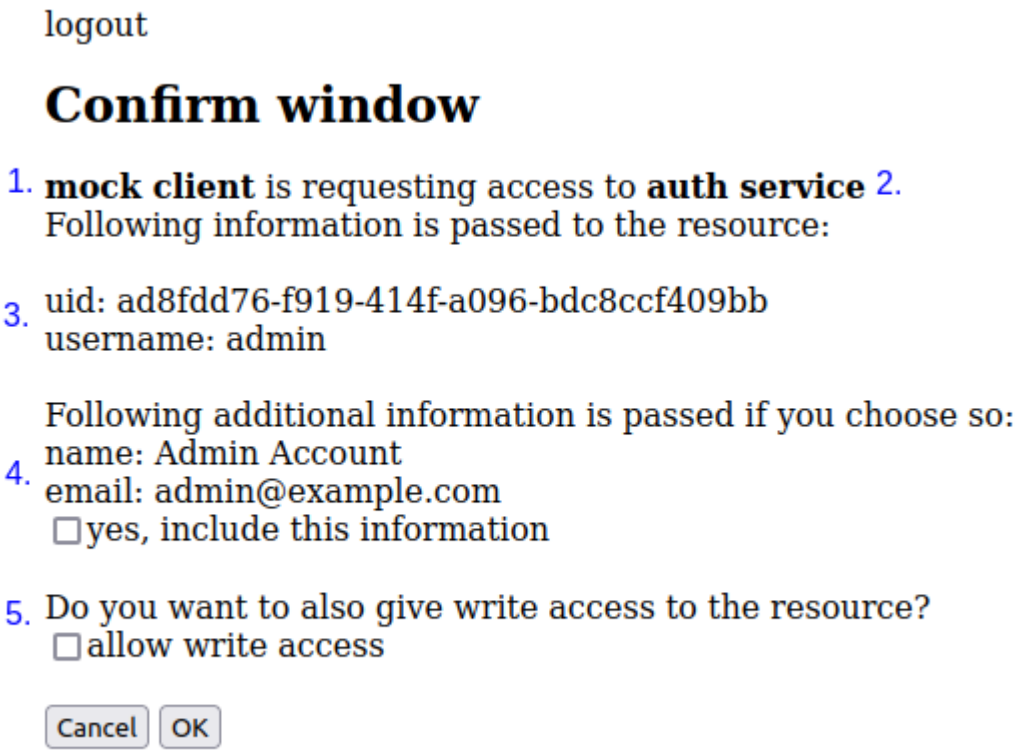
Käyttäjän tunnistautumisen jälkeen valtuutusta pyytäneelle sivustolle välitetään kertakäyttöinen valtuutuskoodi, jolla tämä voi pyytää tokenin resurssin käyttöä varten. Tässä tapauksessa sivustolla viitataan kokonaisuuteen, johon kuuluu myös jonkinlainen palvelinpuolen ohjelmisto. Valtuutuskoodi välitetään sivuston palvelimelle selaimen välityksellä, mutta palvelin hakee tokenin suoraan kirjautumispalvelusta eikä käyttäjän selaimen kautta. Näin token ei ikinä päädy selaimeen eikä näin

ollen voi päätyä myöskään hyökkääjän käsiin esimerkiksi sivuhistorian kautta. Vaikka token itsessään kulkee turvassa, voi hyökkääjä silti päästä käsiksi valtuutuskoodiin ja näin päästä nopeasti hakemaan tokenin ennen valtuutusta pyytänyttä sovellusta. Tältä kuitenkin suojaudutaan siten, että sovellus antaa koodihaasteen (PKCE, kts. kappale 3.2.4), joka on pitkästä satunnaisestä merkkijonosta yksisuuntaisesti luotu tiiviste (eng. hash). Tämä tiiviste tallennetaan kirjautumispalvelussa valtuutuskoodin yhteyteen ja tokenia hakiessa hakijan tulee antaa alkuperäinen satunnainen merkkijono, josta kirjautumispalvelu muodostaa samalla hajautusfunktiolla tiivisteen ja vertaa sitä tallentamaansa tiivisteeseen. Näin valtuutuskoodi yksin ei riitä, vaan hyökkääjän täytyisi arvata myös tuo satunnainen merkkijono kyetäkseen hakemaan tokenin varastamallaan valtuutuskoodilla. (Wilson ja Hingnikar 2019.)

4.4 Palvelun erityispiirteet

RFC 6749 ei ota kantaa muuhun kuin siihen, miten eri osapuolten välinen kommunikointi tapahtuu ja missä järjestyksessä tietoja siirretään. Tämä jättää mahdollisuuden toteuttaa palveluun omia lisäominaisuuksia, kunhan käyttäjän tunnistamiseen ja tokenin välittämiseen liittyvät tapahtumat noudattavat standardia.

Tämän palvelun osalta haluttiin korostaa käyttäjän tietosuojaa. Tästä syystä palvelu varmistaa aina käyttäjältä, mitä tietoja hän haluaa välittää itsestään ja millaisia oikeuksia hän haluaa antaa valtuutusta pyytävälle palvelulle. Valtuutuksen saaneen palvelun tulee toimia niillä tiedoilla ja oikeuksilla, mitä käyttäjä suostuu sille myöntämään, eikä se voi esittää vaatimuksia tarvittavista tiedoista tai oikeuksista. Token sisältää tiedon siitä, mihin resurssiin sen haltijalla on lupa, mutta luonnollisesti tämä edellyttää sitä, että jokainen resurssi tarkistaa nämä oikeudet aina pyynnön yhteydessä.



Kuva 6. Valtuutussivun vahvistusnäkymä

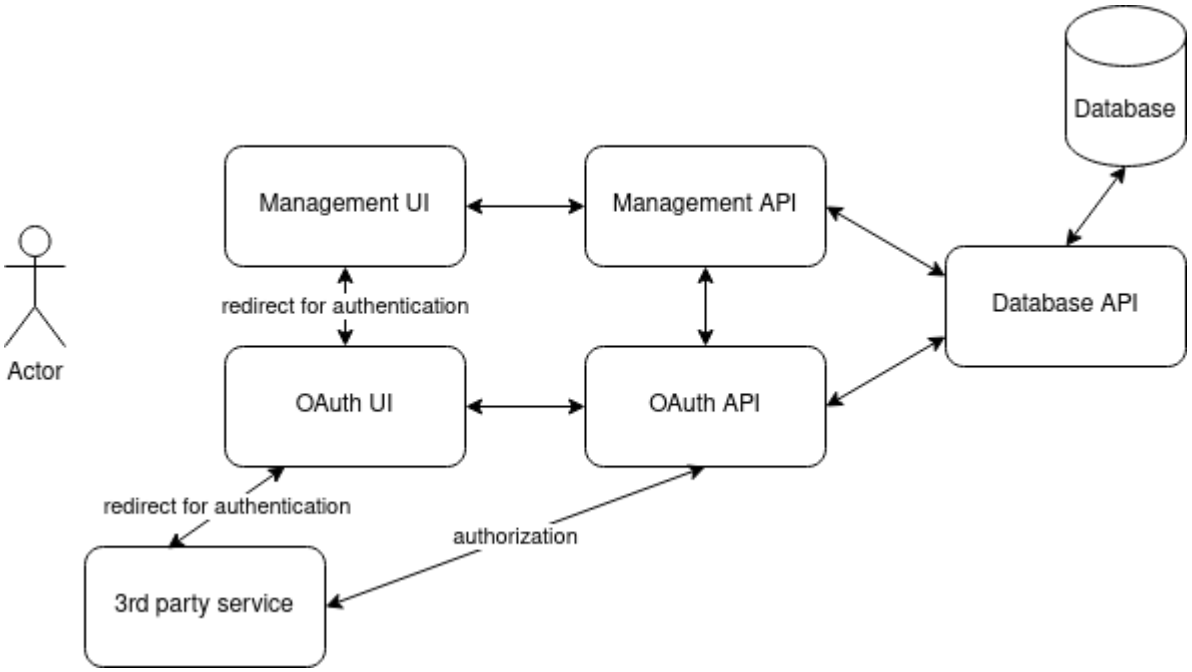
Kuvassa 6. näkyy, miten kohdassa 1. kerrotaan valtuutusta pyytävän sovelluksen nimi ja kohdassa 2. resurssi, johon valtuutusta pyydetään. Tämän jälkeen kerrotaan selkeästi kohdassa 3., mitä tietoja käyttäjästä välitetään hyväksynnän yhteydessä. Käyttäjätunnusta voi halutessaan muuttaa tunnuksensa asetuksista, joten sovellusten ei tule luottaa käyttäjätunnukseen vaan se toimii käytännössä vain ihmisluettavana aliaksena ID:lle. Tämän lisäksi käyttäjä voi valita, mikäli haluaa antaa myös vapaavalintaiset tiedot itsestään kohdassa 4. ja haluaako antaa sovellukselle oikeuden myös muokata ja kirjoittaa resurssiin. Jättämällä kohdan 5. valitsematta, voi käyttäjä antaa pelkän lukuoikeuden resurssiin.

Palvelussa kiinnitettiin myös huomiota käyttäjän oikeuteen tulla unohdetuksi. Tokenia pyytänyt palvelu saa aina tokenin mukana tiedon käyttäjän yksilöivästä tunnisteesta ja käyttäjän sen hetkisen käyttäjänimen, jonka tosin voi vaihtaa halutessaan niin usein kuin haluaa. Lisäksi mukaan voi luovuttaa esimerkiksi nimensä tai sähköpostiosoitteensa näin halutessaan. Jos käyttäjä haluaa, että hänen tietonsa poistetaan, voisi olla työlästä käydä läpi kaikki palvelut, joihin on kirjautunut ja pyytää tietojensa poistoa. Siksi on tärkeää, että tietojen poistamiseksi riittää, kun poistaa tunnuksensa itse kirjautumispalvelusta. Tätä tarvetta varten toteutettiin rajapinta, josta kunkin palvelun tulisi

hakea säännöllisesti tieto poistetuista tunnuksista ja tarvittaessa poistaa tiedot omasta järjestelmästään, mikäli huomaavat jonkun käyttäjänsä tehneen poistopyynnön.

4.5 Mikropalveluiksi pilkkominen

Toteutuksen vaihtaminen valtuutuskoodiin synnytti tarpeen kunnolliselle käyttöliittymälle, jolloin palvelu eriytyi kahteen osaan: käyttöliittymään ja palvelinsovellukseen. Toteutuksen olisi voinut jättää tällaiseksi, mutta siitä haluttiin vielä joustavampi myös ylläpidon ja jatkokehittämisen osalta. Haluttiin rakentaa mikropalveluarkkitehtuuri, joka koostuisi itsenäisistä palveluista, joita voitaisiin kehittää muista riippumattomina. Laadittiin uusi arkkitehtuurikuvaus, jonka laatiminen pohjautuu Atchisonin (2020) viiteen ehtoon itsenäiselle palvelulle. Koska kehitystyöstä vastaa ainakin toistaiseksi yksi henkilö, hallinnon osalta kriteeriksi otettiin mahdollisuus siirtää vastuu yksittäisen mikropalvelun kehityksestä toiselle henkilölle niin haluttaessa.



Kuva 7. Uudessa arkkitehtuurissa itsenäiset palvelut muodostavat kokonaisuuden

Kirjautumispalvelu haluttiin yksinkertaistaa siten, että se huolehtii vain käyttäjän tunnistamisesta ja palveluiden valtuuttamisesta. Tämä tarkoitti sitä, että tarvittiin erillinen palvelu tunnusten hallintaan. Näin palvelinpuolen toteutus pilkottiin kahdeksi kuvassa 7 näkyväksi mikropalveluksi: Management API mahdollistaa käyttäjien lisäämisen, muokkaamisen ja poistamisen, kun taas OAuth API huolehtii todentamiseen ja valtuuttamiseen liittyvistä toiminnallisuuksista. Koska määritelmämme mukaan mikropalveluiden täytyy omistaa oma datansa, eivät nämä kaksi erillistä palvelua voineet

jakaa tietokantaa keskenään. Tästä syystä tietokantapalvelut päätettiin irrottaa täysin omaksi palvelukseen, jota käytettäisiin REST-rajapinnan kautta sisäisessä verkossa, jotta tietokantaan ei pääsisi käsiksi suoraan avoimesta internetistä. Kuvassa 7 tietokantapalvelut näkyvät nimellä Database API. Nämä kolme API-palvelua tuottivat palvelinpuolen toiminnallisuuden.

Palvelinpuolen lisäksi myös käyttöliittymä jaettiin kahdeksi erilliseksi osaksi. Toinen vastasi pelkästään käyttäjän todentamisesta ja valtuutusten myöntämisestä ja toinen puolestaan huolehti käyttäjätunnuksen hallinnoimisesta. Ensin mainittu on merkitty kuvassa 7 nimellä OAuth UI ja jälkimmäinen nimellä Management UI. Nämä kaksi käyttöliittymää olivat käyttäjälle näkyvä rajapinta palveluun ja ne kommunikoivat kumpikin oman palvelinpuolen rajapintansa kautta tietokantapalvelun kanssa.

Kun ulkopuolinen sivusto halusi valtuutuksen johonkin palveluun, se ohjasi käyttäjän kirjautumisesta vastaavaan käyttöliittymään, jossa käyttäjä tunnistautui ja sai valtuutuskoodin. Tämän koodin käyttäjä toi mukanaan takaisin sivustolle, joka sitten pystyi koodia vastaan hakemaan tokenin kirjautumisista ja valtuutuksista huolehtivalta API:lta. Tämän jälkeen sivusto pystyi käyttämään tokenin avulla niitä resursseja, joihin sille oli myönnetty valtuutus. Valtuutus puolestaan tarkistettiin valtuutuksista huolehtivalta API:lta.

Jos käyttäjä puolestaan halusi hallinnoida tunnustaan ja esimerkiksi vaihtaa salasanansa tai muita tietojaan, kuten sähköpostiosoitetta tai käyttäjätunnusta, hän avasi hallintakäyttöliittymän. Tämä käyttöliittymä ohjasi käyttäjän kirjautumisesta vastaavaan käyttöliittymään, joka palautti käyttäjän valtuutuskoodin kanssa takaisin hallintakäyttöliittymään. Hallintakäyttöliittymä välitti valtuutuskoodin edelleen omalle hallinta API:lleen, joka haki sen avulla tokenin, jolla sai valtuutuksen käyttää tietokantarajapintaa.

Huomionarvoista tässä toteutuksessa on, että tilatietoja ei säilytetä missään ja pysyvää dataa säilyttää yksi ainoa tietokanta. Käytännössä esimerkiksi kuvan OAuth API olisi voitu toteuttaa tilallisena, jossa se olisi säilyttänyt väliaikaisesti haastekoodia ja muita tunnistautumis- ja valtuutusprosessiin tarvittavia tietoja, mutta tämä olisi voinut aiheuttaa myöhemmin skaalautuvuusongelmia. Tilattomana OAuth API ja Management API on mahdollista suorittaa useina rinnakkaisina toisistaan tietämättöminä prosesseina eikä käyttäjän kannalta ole merkitystä, mikä prosessi milloinkin vastaa hänen pyyntöönsä, koska prosessit eivät säilytä tilatietoa. Toisin sanoen palveluja voidaan skaalata horisontaalisesti ja vain siltä osin, kuin on tarvetta. Myöskään Database API ei säilytä minkäänlaista tilatietoa itsessään, joten sekin olisi mahdollista skaalata horisontaalisesti, jolloin useampi prosessi tarjoaisi rajapintaa varsinaiseen tietokantainstanssiin.

Arkkitehtuuri mahdollistaa myös myöhemmin ketterät muutokset toteutukseen. Esimerkiksi tietokantaratkaisun vaihtaminen vaikuttaa vain Database API:n toimintaan, eikä näin ollen vaadi minkäänlaisia muutoksia kokonaisuuden muihin osiin. Samalla tavoin valtuutustoiminnallisuuteen on mahdollista lisätä uusia turvallisuutta parantavia ominaisuuksia, kuten vaihtaa tokenien allekirjoitusmenetelmää, koskematta muihin palvelun osiin. Tämä tekee ylläpidosta ja jatkokehityksestä helpompaa, kun tarvitsee muuttaa vain yhtä osaa kerrallaan. Laajimmillaankin esimerkiksi uuden tiedon tallentaminen käyttäjästä Management API:n puolella vaatii vain pienen päivityksen Database API:n puolella ja nämäkin muutokset voidaan toteuttaa eri aikaan, jolloin uusi toiminnallisuus alkaa toimi sitten, kun molemmat palvelut ovat toteuttaneet vaadittavat muutokset.

5 Pohdinta

5.1 Tavoitteissa onnistuminen

Palvelua lähdettiin alun perin toteuttamaan melko hataralla pohjatyöllä, jolloin osittain sattumalta päädyttiin toteuttamaan OAuth 2.0 kehyksen valtuutusmallia, joka perustuu käyttäjän tunnistetietoihin. Ratkaisu itsessään oli hyvin yksinkertainen ja helppo toteuttaa ja etenkin sen käyttöönotto kirjautumista vaativassa palvelussa olisi ollut hyvin suoraviivaista. Ratkaisu oli kuitenkin kömpelö ja sen todettiin olevan tietoturvan näkökulmasta ongelmallinen. Se ei myöskään ollut todellinen kertakirjautuminen. Aito kertakirjautuminen otettiin kuitenkin vaatimukseksi kehitystyön edetessä. Alkuperäisissä tavoitteissa myös korostettiin käyttäjän oikeutta valita hänestä luovutettavat tiedot. Tässä alkuperäisessä toteutuksessa jouduttiin kuitenkin myös tämän osalta tekemään kompromissi, jossa käyttäjä voi ennalta määritellä hänestä luovutettavat tiedot, mutta ei tapauskohtaisesti yksittäisen palvelun osalta.

Haasteiden takia päädyttiin tutkimaan perusteellisemmin vaihtoehtoja, joita palvelun toteuttamiseen olisi tarjolla. Ratkaisuksi valikoitui OAuth 2.0 kehyksen valtuutuskoodiin perustuva toteutus. Siinä käyttäjä pystyy jokaisen valtuutuksen yhteydessä valitsemaan mitä tietoja hänestä välitetään valtuutuksen pyytäjälle ja lisäksi käyttäjä voi valikoida millaisia valtuutuksia myönnetään. Valtuutus voidaan lisäksi myöntää asiakas- ja resurssikohtaiseksi, joten saamalla valtuutuksen yhteen resurssiin, ei automaattisesti saa valtuutusta muihin resursseihin. Tämä vähintäänkin kattaa vaatimuksen siitä, että käyttäjä voi kontrolloida mitä tietoja hänestä luovutetaan ja kenelle niitä luovutetaan. Myös vaatimus mahdollisuudesta tietojen poistoon saatiin toteutettua jollakin tasolla, vaikka toteutus ei olekaan optimaalinen. Tässä vaiheessa tietojen poistaminen perustuu asiakassovellusten aktiivisuuteen, sillä näiden täytyy aktiivisesti itse hakea tietoa poistopyynnöistä ja tämän jälkeen poistaa tiedot. Jossain tilanteessa voi kuitenkin olla, ettei asiakassovellus huolehdi tästä velvollisuudestaan ja tiedot jäävät sovelluksen tietokantaan.

Keskeinen tavoite helposta ja vaivattomasta kirjautumisesta useisiin palveluihin yhden tunnuksen avulla toteutui sulavasti. Tähän auttoi merkittävästi siirtyminen valtuutuskoodiin perustuvaan tekniikkaan, jolloin kirjautuminen tapahtuu turvallisesti vain ja ainoastaan kirjautumispalvelimen kanssa ja se täytyy tehdä vain kerran. Toisin sanoen yhdellä kirjautumisella saa kaikki toteutettua kirjautumistapaa käyttävät palvelut käyttöönsä. Tässä yhteydessä kuitenkin täytyy todeta, että refresh tokenin puute syö jonkin verran käyttäjäkokemusta, sillä rajatulla voimassaololla varustetut tokenit vanhenevat ja ainoa tapa saada uusi, on pyytää sellaista käyttäjältä. Toisin sanoen täydellisesti tavoitetta vastaava palvelu vaatisi refresh tokenin implementoinnin.

Tietosuojan lisäksi haluttiin kiinnittää huomiota tietoturvaan. Tietoturvan näkökulmasta ratkaisu on onnistunut, sillä siinä on sisäänrakennettuja suojamekanismeja tyypillisimpiä hyökkäyksiä vastaan. Kaikki pyynnöt palvelinpuolelle tulee etukäteen määritellystä käyttöliittymästä, jolloin selaimet sulkevat pois erilaiset cross site scripting -hyökkäykset. Toteutus pohjaa vahvasti selaimen uudelleenohjaukseen, jolloin pahantahtoinen sivusto voisi yrittää uudelleenohjauksella aiheuttaa vahinkoa käyttäjälle. Tältäkin kuitenkin on suojauduttu vaatimalla, että jokainen asiakassovellus tulee olla rekisteröity kirjautumispalvelimella ja uudelleenohjauspyyntöjä sallitaan vain hyväksyttyihin osoitteisiin. Lisäksi näiden ohjauksien toimiminen edellyttää state-parametrin käyttö, jonka avulla pyritään estämään cross site request forgery -hyökkäyksiä. Tietoturvaan on panostettu myös vaatimalla PKCE:n käyttöä ja toteuttamalla kaikki liikenne salattua HTTPS-protokollaa käyttäen.

Mikropalveluarkkitehtuuriksi pilkkominen mahdollisti joustavan kehitystyön tarjoamalla mahdollisuuden kehittää yhtä ominaisuutta kerrallaan ilman riippuvuuksia muiden palvelun osien koodiin. Tämä kuitenkin edellytti huolellista rajapintakuvausten laatimista ja niiden noudattamista, sillä eri mikropalvelut kommunikoivat näiden rajapintojen kautta keskenään. Tilattomien pienten mikropalveluiden skaalaaminen on myös melko suoraviivainen toimenpide, joten tarvittaessa palvelukokonaisuuden suorituskykyä on helppo sovittaa vastaamaan käyttöä.

Palveluun on helppo kiinnittää uusia asiakassovelluksia, jolloin kaikki kirjautumispalvelussa olevat tunnukset toimivat välittömästi myös uuden palvelun kanssa. Koska kaikki tunnistautumiset tehdään kirjautumispalvelussa, on salasana aina ajan tasalla, eikä ole riskiä esimerkiksi vanhentuneista välimuisteista. Tavoitteissa on siis onnistuttu melko hyvin ja muutaman pienen jatkokehitysvaiheen kautta palvelu toteuttaisi kirkkaasti asetetut tavoitteet ja jopa paljon enemmän.

5.2 Hyödynnettävyys

Toteutus on erittäin käyttökelpoinen ja tarpeellinen erilaisten palvelukokonaisuuksien kanssa. Vaikka tässä yhteydessä on keskitytty vain toisistaan irrallisiin web-sivustoihin, ei mikään estä käyttämästä pohjateknologiaa myös muissa yhteyksissä, kuten mobiilisovelluksissa. Halutessa valtuutuskoodin antaminen voidaan tehdä automaattisten uudelleenohjausten välityksellä siten, ettei kirjautuneelta käyttäjältä pysähdytä kysymään lupaa, mikäli valtuutusta pyytävät palvelut ovat luotettavia. Näin on mahdollista toteuttaa laajan kokonaisuuden käyttäjähallinta siten, että kerran kirjauduttuaan asiakas pääsee käsiksi esimerkiksi pankki- ja vakuutuspalveluihin, vaikka nämä palvelut olisi toteutettu omina toisista tietämättöminä palasina. Toisena esimerkkinä asiakas voi päästä samalla kirjautumisella ajanvaraukseen ja katsomaan laboratoriotuloksiaan, vaikka nämä kaksi toimintoa olisivat täysin toisistaan riippumattomat.

Ratkaisu on RFC 6749 mukainen ja se on turvallisena pidetty standardi. Tässä toteutuksessa on kiinnitetty erityistä huomiota käyttäjän oikeuteen hallita omia tietojaan ja erityisesti käyttäjän oikeutta tulla unohdetuksi. Tämänkaltainen ratkaisu voisi toimia esimerkiksi monissa verkkoyhteisöissä, joissa eri aihealueet ovat jakautuneet omille alustoilleen, mutta sama tunnus toimii kaikkiin alustoihin. Halutessaan käyttäjä voi valita, millä tiedoilla hän näkyy milläkin alustalla ja toisaalta hän pystyy helposti yhdellä pyynnöllä poistamaan itsensä kaikilta alustoilta.

Kuvatut käyttötapaukset ovat arkipäivää ja näin ollen toteutus on ajankohtainen jo tänä päivänä. Mikropalveluita käsittelevässä osassa todettiin Atchisoniin (2020) vedoten, että palveluiden toteutuksessa trendi on kohti yhä pienempiä ja pienempiä palveluita. Näin ollen käyttäjän saumaton siirtymä palvelukomponenttien välillä on ehdottoman tärkeää käyttäjäkokemuksen näkökulmasta. Toteutettu ratkaisu tarjoaa mahdollisuuden tähän.

5.3 Jatkokehityskohteita

Toteutettu ratkaisu on toimiva, mutta siinä on omat puutteensa. Osa jatkokehityskohteista pyrkii korjaamaan näitä puutteita ja toiset puolestaan pyrkivät lisäämään tietoturvaa tai käytettävyyttä. Lisäksi pohditaan mahdollisuutta parantaa palvelun suorituskykyä ottamalla käyttöön uusia teknologioita.

5.3.1 Käyttäjätunnistuksen parannukset

Tällä hetkellä käyttäjältä vaaditaan käyttäjätunnuksen ja salasanan muistamista kirjautuessa ja samalla tämä on ainoa tunnistautumistapa käyttäjälle. Melko yksinkertainen parannus tilanteeseen olisi tarjota kaksivaiheista kirjautumista sähköpostin avulla, jolloin tietoturva parantuisi välittömästi asteen paremmaksi. Toisaalta mielenkiintoinen vaihtoehto olisi tutkia myös salasanatonta kirjautumista sähköpostin avulla siten, että käyttäjä antaisi kirjautuessa vain esimerkiksi sähköpostiosoitteensa, minkä jälkeen hänelle lähetettäisiin sähköpostilla kertakäyttöinen koodi, jolla kirjautuminen onnistuu rajatun ajan. Näin käyttäjän ei välttämättä tarvitsisi muistaa yhtä ainoaa salasanaa, vaan kirjautuminen onnistuisi muistamalla pelkkä sähköpostiosoite.

Toteutusta täytyisi tutkia huolella, sillä salasanattomassa kirjautumisessa täytyy jotenkin estää pahantahtoista tahoa syöttämästä toistuvasti saman käyttäjän sähköpostia aiheuttaen tälle sähköpostitulvan. Lisäksi esimerkiksi lyhyen numerokoodin arvaaminen on helpompaa kuin hyvän salasanan, vaikka numero olisikin voimassa vain rajatun ajan. Vähintäänkin täytyisi varmistaa annetun koodin satunnaisuus ja riittävä pituus, jotta sen arvaaminen olisi mahdollisimman vaikeaa.

5.3.2 Valtuutuksen säilyttäminen

Palvelu ei tällä hetkellä anna lainkaan refresh tokenia token-pyynnön yhteydessä. Tästä johtuen sovellus saa vain rajallisen aikaa pääsyn pyytämäänsä resurssiin, minkä jälkeen se joutuu pyytämään uutta valtuutuskoodia uuden tokenin hakemiseen. Tällainen tilanne voi pahimmassa tapauksessa olla hyvin hankala käsitellä valtuutusta tarvitsevan sovelluksen toimesta ja se aiheuttaa huonon käyttäjäkokemuksen. Palveluun tulisi ehdottomasti toteuttaa refresh token -toiminnallisuus, joka mahdollistaisi sovellusten pyytää uutta tokenia taustalla vaivaamatta käyttäjää. Tämän yhteydessä tulisi samalla ruveta pitämään kirjaa, mille sovelluksille ja mihin resursseihin käyttäjä on oikeuksia myöntänyt, jotta käyttäjä voisi halutessaan perua tokenin ja siihen liittyvän refresh tokenin.

Valtuutuksen säilymisen tärkeys korostuu tilanteessa, jossa sovelluksen täytyy päästä resurssiin kiinni myös silloin, kun käyttäjä ei ole aktiivisesti sovelluksen äärellä. Näin voi olla esimerkiksi silloin, kun halutaan tehdä raskaita ajoja hiljaiseen aikaan tai mahdollistaa käyttäjälle ajastettuja toimintoja.

5.3.3 Epäsymmetrinen allekirjoitus

Myös tokenien allekirjoitustekniikka tulisi ottaa syvempään tarkasteluun. Tällä hetkellä tokenit allekirjoitetaan kirjautumispalveluun määritellyllä avaimella. Ainoa tapa eri resursseilla varmentaa tokenin allekirjoitus on lähettää token kirjautumispalvelulle ja pyytää vahvistamaan token. Tämä ei ole ihanteellista, vaan parempi ratkaisu olisi käyttää epäsymmetristä salausta, jossa kirjautumispalvelu allekirjoittaisi tokeni omalla yksityisellä avaimellaan ja antaisin tokenissa tiedon, mistä julkinen avain allekirjoituksen tarkistukseen löytyy. Tällöin resurssi voisi tallentaa kirjautumispalvelun julkisen avaimen tilapäisesti itselleen ja tarkistaa tokeneita sen avulla ilman tarvetta tehdä jokaisesta tokenista erillistä pyyntöä kirjautumispalveluun.

Epäsymmetrinen salaus tokeneissa on kuitenkin haastava toteutettava, koska huonosti toteutettuna se tarjoaa mahdollisuuden väärentää allekirjoituksia. Jos resurssi ei tarkista julkisen avaimen lähdettä, voi pahimmassa tapauksessa hyökkääjä allekirjoittaa omilla avaimillaan tokeneita ja ohjata niitä vastaanottavan palvelun oman julkisen avaimensa ääreen. Epäsymmetrinen salaus tarjoaa siis parannuksia vähentämällä liikennettä ja palvelinten kuormitusta, mutta vaatii huolellisuutta toteutuksessa, jotta siitä ei synny vakavaa tietoturvaongelmaa.

5.3.4 Palveluiden välinen kommunikaatio

Yksi varteenotettava jatkokehityskohde on palveluiden välisen liikenteen keventäminen. Nykyisessä toteutuksessa OAuth API, Management API ja Database API keskustelevat toistensa kanssa välittäen tietoa JSON-muodossa REST-rajapintojen kautta. Tämä tapahtuu HTTP/1.1-

protokollaa käyttäen eikä ole välttämättä tehokkain ratkaisu palvelukokonaisuuden sisäiseen liikenteeseen. Tässä olisi syytä selvittää gRPC-kehyksen käyttöönottoa sisäisessä liikenteessä ja tutkia, parantaako se palvelukokonaisuuden suorituskykyä. Mahdollisuuksien mukaan muitakin ratkaisuja mikropalveluiden väliseen kommunikaatioon voi tutkia, mikäli ne tarjoavat etua REST-rajapintaan nähden.

Erityisesti voisi ajatella, että kommunikaatio Database API:n suuntaan voitaisiin toteuttaa tehokkaammin jollakin muulla kuin REST-rajapinnalla, sillä palvelua käytetään vain sisäisesti ja se käytännössä toimii vain abstraktiotasona varsinaiselle tietokannalle. Pääsy tietokantaan täytyy kuitenkin olla nopeaa, koska tunnistamiseen ja valtuuttamiseen liittyvä prosessi yksinään joutuu tekemään suuren määrän tietokantakyselyitä ja toteutustavasta riippuen myös resurssien tekemät tokenin vahvistuspyynnöt voivat vaatia tietokantakyselyä.

5.4 Ammatillinen kehittyminen

Projektiin lähdettiin lopulta heikohkolla valmistautumisella. Tästä seurasi, että tavoitteet muuttuivat projektin aikana, kun alkuperäinen idea toteutuksesta paljastui puutteelliseksi. Keskeinen havainto oli, että huolellisella valmistautumisella olisi ollut mahdollista säästää monta tuntia aikaa ohjelmointityöstä ja dokumentaation kirjoittamisesta. Toisaalta voi olla vaikeaa tietää etukäteen täysin, mitä tuotteelta halutaan, koska voi olla puutteita ja tarpeita, jotka paljastuvat vasta kokeiltaessa. Näin voi myös ajatella, että vaatimusten täsmentyminen ja arkkitehtuurin muutokset projektin aikana opettivat, miten sopeuttaa olemassa oleva toteutus muuttuneisiin tarpeisiin.

Toteutuksen aikana löytyi lukuisia RFC-dokumentteja, joihin perehtyminen oli opettavaista jo itsessään. Niiden lukeminen ja ohjelmistotuotteen valmistaminen näiden annettujen standardien mukaisesti oli uudenlainen kokemus, jonka uskaltaa väittää olleen myös opettavainen kokemus ammatillisen kehittymisen näkökulmasta. Näin kahdestakin eri näkökulmasta, joista ensimmäinen näkökulma on se, että tuote täytyi saada toteutettua siten, että se noudattaa annettua standardia. Työtä ei siis voinut tehdä aivan villisti vaan piti varmistaa, että se täyttää kaikki annetut kriteerit. Toisaalta jatkossa erilaisia ratkaisuja toteuttaessa tulee varmasti aktiivisemmin etsittyä standardeja, jotka voisivat tarjota mallin asian toteuttamiseen, jolloin saadaan turvallinen ja toimivaksi todettu ratkaisu ilman tarvetta keksiä kaikkea itse uudestaan.

Projektin aikana ymmärrys mikropalveluista on kasvanut ja erilaisten palveluarkkitehtuurien miettiminen on luontevampaa ja sujuvampaa kuin aiemmin. Sama koskee tietoturvaa, jonka osalta perspektiivi on laajentunut runsaasti projektia toteutettaessa. Sekvenssikaavioiden piirtäminen on myös osoittautunut käteväksi työkaluksi hahmottaa sovelluksen toimintaa ja eri komponenttien välistä vuorovaikutusta.

Lähteet

Atchison, L. 2020. Architecting for scale. O'Reilly Media. Sebastopol. E-kirja. Luettu 5.10.2022.

Fowler, M. 2013. UserStory. Luettavissa: https://martinfowler.com/bliki/UserStory.html. Luettu 20.11.2022.

Fowler, M. & Lewis, J. 2014. Microservices. Luettavissa: https://martinfowler.com/articles/microservices.html. Luettu 26.6.2022.

Internet Engineering Task Force 2012. The OAuth 2.0 Authorization Framework. Luettavissa: https://datatracker.ietf.org/doc/html/rfc6749. Luettu 26.6.2022.

Internet Engineering Task Force 2015a. JSON Web Token (JWT). Luettavissa: https://datatracker.ietf.org/doc/html/rfc7519. Luettu 16.11.2022.

Internet Engineering Task Force 2015b. Proof Key for Code Exchange by OAuth Public Clients. Luettavissa: https://datatracker.ietf.org/doc/html/rfc7636. Luettu 14.11.2022.

Isaacson, C. 2014. Understanding Big Data Scalability: Big Data Scalability Series, Part I. Pearson. New Jersey. E-kirja. Luettu 20.11.2022.

Larrucea, X., Santamaria, I., Colomo-Palacios, R. & Ebert, C. 2018. Microservices. IEEE software 2018, vol. 35, s. 96-100. Luettavissa: https://ieeexplore.ieee.org/document/8354423. Luettu 10.7.2022.

MDN Web Docs. 2022a. Cross-Origin Resource Sharing (CORS). Luettavissa: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. Luettu 10.7.2022.

MDN Web Docs. 2022b. Window.localStorage. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage. Luettu 30.10.2022.

Nadareishvili, I., Mitra, R., McLarty, M. & Amundsen, M. 2016. Microservice architecture - aligning principles, practices, and culture. 1. painos. O'Reilly Media. Sebastopol. E-kirja. Luettu 25.6.2022.

Newman, S. 2019. Monolith to Microservices. O'Reilly Media. Sebastopol. E-kirja. Luettu 1.7.2022.

NIIT, 2002. Special Edition Using TCP/IP, Second Edition. Que Publishing. E-kirja. Luettu 4.12.2022.

RFC Editor s.a. Official Internet Protocol Standards. Luettavissa: https://www.rfc-editor.org/standards. Luettu: 4.12.2022

Sun, S.-T., Pospisil, E., Muslukhov, I., Dindar, N., Hawkey, K. & Beznosov, K. 2013. Investigating Users' Perspectives of Web Single Sign-On: Conceptual Gaps and Acceptance Model. ACM transactions on Internet technology, Vol. 13 (1), s. 1-35.

Taibi, D., Lenarduzzi, V., Pahl, C. & Janes, A. 2017. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. 2017 XP Scientific Workshops, XP 2017, Cologne, s. 1-5.

Wilson, Y. & Hingnikar, A. 2019. Solving Identity Management in Modern Applications: Demystifying OAuth 2.0, OpenID Connect, and SAML 2.0. Apress. New York. E-kirja. Luettu 2.7.2022.