REST-pohjaisten web-sovellusten autentikaatio ja turvallisuus

Juho Sirén

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Huhtikuu 2016

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelytieteiden tutkinto-ohjelma
REST-pohjaisten web-sovellusten autentikaatio ja turvallisuus
Juho Sirén
Pro gradu -tutkielma, 51 sivua
Huhtikuu 2016

Yksinkertaisen ja turvallisen rajapinnan toteuttaminen mahdollisimman montaa erilaista asiakassovellusta ja -laitetta palvelemaan on ongelma, joka tulee ratkaista, jotta verkkoon tietonsa tallentavien sovellusten rakentaminen on mahdollisimman tehokasta. Representational state transfer, eli lyhemmin REST, on arkkitehtuurimalli, jonka avulla voidaan luoda HTTP-protokollaan perustuva helposti laajennettava palvelinpuolen sovellus, joka pyrkii tarjoamaan mahdollisimman yhdenmukaisen rajapinnan.

Tässä tutkielmassa tutkitaan, kuinka REST:n käyttäminen vaikuttaa internetsovelluksen tietoturvaan ja kuinka REST-pohjaisessa sovelluksessa käyttäjä voidaan tunnistaa. Tutkielmassa analysoidaan kymmentä yleisintä tietoturvauhkaa sekä kahdeksaa erilaista tunnistamiskeinoa, joiden kaikkien toimintaan perehdytään yhdessä REST-pohjaisten sovellusten kanssa.

Tutkielman perusteella voidaan todeta, että REST:n valitseminen ei yksistään takaa turvallista sovellusta, mutta sen käytäntöjen noudattaminen tekee sovelluksen testaamisesta, laajentamisesta sekä ylläpitämisestä yksinkertaisempaa ja siten myös tietoturvallisempaa. REST:n määrittämät rajoitukset ja sen tuottamat hyödyt myös vaikuttavat osaltaan autentikaatiokeinon valintaan sekä turvallisuusuhkien torjuntaan.

Avainsanat ja -sanonnat: REST-rajapinta, todentaminen, tietoturva

Sisällys

1. Johdanto

Internet ei ole enää vuosiin rajoittunut pöytätietokoneiden käyttöön, vaan mobiiliverkkoyhteydet ja aina vain pienemmät laitteistokomponentit mahdollistavat verkkoyhteyden käyttämisen yhä useammalla laitteella ja lähes paikassa kuin paikassa. Vaikka verkkosovellus onkin usein riippuvainen datasta, datan tulee olla mahdollisimman itsenäistä, eikä sen tule ottaa kantaa siihen, kuinka ja missä sitä käytetään.

Sama tieto saattaa olla käytössä sekä matkapuhelimissa, älykelloissa että kannettavissa tietokoneissa, minkä takia ei välttämättä riitä, että yksittäinen asiakassovellus pyrkii taipumaan mahdollisimman moneen näyttöön. Samaa dataa hyödyntäen voidaan kehittää eri laitteille web-sovellusten rinnalle kustomoituja natiivisovelluksia, jotka ovat suoritusteholtaan ja samalla myös käyttökokemukseltaan parempia kuin esimerkiksi HTML:llä ja JavaScriptillä toteutetut verkkosivut.

Koska datan monistaminen ei ole kannattavaa, on datan tarjoaminen järkevintä yhdestä yhdenmukaisesta rajapinnasta, joka on toteutettu niin, että mahdollisimman moni laite ja sovellus voi sitä hyödyntää. Datan hyödyntämistä helpottaa kattavaakin dokumentaatiota paremmin standardeita noudattava ja yhdenmukainen rajapinta.

Datan itseisarvo on usein sen yksityisyys, mutta myös mahdollisuus jakaa se kaikille niille, jotka sitä tarvitsevat. Rajapintaa rakennettaessa kehittäjän tulee lisäksi ottaa huomioon myös käyttäjä ja tämän yksityisyys ja tietoturvallisuus. Rajapinta on onnistunut vasta sitten, kun käyttäjä ja kehittäjä voivat olla varmoja, että tallennettua tietoa pääsee näkemään ja muokkaamaan vain sellaiset ihmiset, joille pääsy kuuluu.

Representational State Transfer, eli REST, on Roy Fieldingin jo vuonna 2000 esittelemä arkkitehtuurimalli, joka pyrkii vastaamaan palvelimen haasteisiin lukemattomien erilaisten asiakassovellusten ja -laitteiden edessä. Se määrittelee yhdenmukaisen, kerrostetun ja mahdollisimman helposti ylläpidettävän rajapinnan datalle eli resursseille sekä dataan liittyville ominaisuuksille eli resurssien tilalle.

Tässä tutkielmassa tutustutaan internetpalveluiden tietoturvaan ja käyttäjän todentamiseen sekä analysoidaan eri todentamistapojen hyödyntämistä ja yleisimpiä turvallisuusuhkien ehkäisyä REST-pohjaisia järjestelmiä suunniteltaessa. REST:n kannalta oleellisimpia autentikaatiotapoja ja turvallisuusuhkien estämistä demonstroidaan tutkielmaa varten Node.js:n ja Sails.js-ohjelmistokehyksen avulla toteutetulla REST-rajapinnan tarjoavalla verkkosovelluksella sekä rajapintaa käyttävällä Angular 2 -asiakassovelluksella.

2. Web-sovellusten turvallisuus

Web-sovellusten turvallisuus on usean tekijän summa ja pahimmillaan merkittävän turvallisuusuhan aiheuttaa yksittäinen huolimattomasti toteutettu komponentti tai rajapinta. Turvallisuutta voidaan parantaa huolellisella suunnittelulla, testauksella, noudattamalla hyväksi todettuja käytäntöjä sekä käyttämällä tekniikoita, joiden turvallisuuden on varmistanut mahdollisimman suuri kehittäjäjoukko.

Tässä tutkielmassa esitellään The Open Web Application Security Foundationin vuonna 2013 tuottama OWASP Top 10 - 2013 The Ten Most Critical Web Application Security Risks -lista, jossa on käyty läpi nimensä mukaisesti kymmenen vuonna 2013 kriittisimmäksi koetuinta verkkosivustoihin kohdistunutta hyökkäystä sekä esitellään keinot, joilla hyökkäyksiä voidaan ehkäistä.

2.1. Cross-site scripting

Cross-site scripting, lyhemmin XSS tai joskus myös CSS, on dynaamisiin verkkosivuihin liittyvä turvallisuusuhka, jossa käyttäjän tuottamaan sisältöön sisällytetään ohjelmakoodia, jonka selain suorittaa sisältöä näyttäessään. Koska onnistunut XSS-hyökkäys suoritetaan nimenomaan uhrin omalla tietokoneella, koskee uhka mitä tahansa laitteelle tallennettua lokaalia tietoa, kuten evästeisiin tallennettuja käyttäjätunnus-salasana-pareja ja asemalle tallennettuja tiedostoja. Onnistunut XSS-hyökkäys voi esimerkiksi vuotaa henkilökohtaisia tietoja verkkoon tai salata tietokoneen kovalevyn ja kiristää käyttäjää levyn salauksen purkamiseksi. [Cross, 2007]

Käyttäjä voi itse suojautua XSS-hyökkäykseltä estämällä selaimensa asetuksista skriptien suorittamisen, mutta koska varsinkin JavaScript on hyvin yleinen tapa esittää sisältöä ja vahvistaa sovelluksen vuorovaikutusta, jää XSS-hyökkäysten estäminen viimekädessä verkkopalvelun tuottajan vastuulle [Cross, 2007]. Kehittäjä voi varautua XSS-hyökkäyksiin esimerkiksi suodattamalla käyttäjien tuottamasta sisällöstä kaikki selaimen suorittamiin skripteihin liittyvät tunnisteet, kuten JavaScript-sisältöön liittyvät <script>-tunnisteet. Sisältö voidaan suodattaa joko ennen sisällön tallentamista tietokantaan tai sen jälkeen, kun tieto haettu tietokannasta ja sitä ollaan esittämässä muille käyttäjille [OWASP, 2013].

2.2. Injektiot

Injektiolla tarkoitetaan hyökkäystä, jossa hyökkääjä pyrkii manipuloimaan tietokannan sisältöä syöttämällä palvelimelle lähetetyn syötteen mukana tietokantaa ohjaavia komentoja, kuten poisto-, haku-, muokkaus- ja lisäyskäskyjä [OWASP, 2013]. Onnistuneella injektiolla hyökkääjä voi esimerkiksi saada haltuunsa kirjautumiseen tarkoitetun tekstikentän kautta järjestelmänvalvojan tunnukset ja lopulta kirjautua saaduilla tunnuksilla sisään. Injektiosta tunnetuin on SQL-tietokantoihin suunniteltu SQL-injektio.

Jos SQL-tietokantaa hyödyntävään Node.js-sivustoon kirjautuminen tapahtuu käyttäjätunnuksen ja salasanan avulla, voisi käyttäjän todentamisen toteuttaa esimerkiksi koodikatkelmassa 1 esitellyllä tavalla. Tällöin onnistuneeksi SQL-injektioksi riittäisi käyttäjätunnus ”;DROP TABLE Users”, jolloin SQL-palvelin suorittaisi komennon ”SELECT * FROM Users WHERE username = ''; DROP TABLE Users;”, joka poistaisi tietokannasta käyttäjiä listaavan Users-taulun.

```
1
2  var username = req.body.username;
3  var password = req.body.password;
4
5  loginSQLCommand = "SELECT * FROM Users WHERE username ='" + username + "' AND password ='" + password + "'";
6
```

Koodikatkelma 1: Kirjautuminen SQL:n ja Node.js:n avulla

Tehokkain tapa estää SQL-injektio on välttää syötettävien arvojen sijoittamista suoraan SQL-lauseeseen ja käyttää syötteiden suodattamiseksi turvalliseksi ja toimivaksi testattua kirjastoa. Kirjastot tunnistavat syötteestä SQL:n kannalta kriittiset erikoismerkit ja merkkaavat ne niin, etteivät ne pysty suoraan toimimaan SQL-komentoina. Esimerkiksi SQL-injektioiden ehkäisemiseksi erikoismerkkien eteen lisätään usein \-merkki. [OWASP, 2013]

2.3. Virheellinen todentaminen ja istuntojen hallinta

Sovelluksen turvallisuuteen liittyy olennaisesti istuntojen ja käyttäjätodentamisen toteutus. Huoleton toteutus voi helposti altistaa sovelluksen lukuisille erilaisille turvallisuusuhkille, kuten istuntojen tai käyttäjätunnusten väärinkäytölle. Toteutuksessa tulee ottaa huomioon erilaiset käyttäjästä johtuvat ja johtumattomat skenaariot. Jos esimerkiksi istunnon tunniste sisällytetään URL:iin, riittää pelkkä URL:n kopioiminen istunnon väärinkäyttöön. Jos sovellusta, jonka istunto ei koskaan vanhene ja käyttäjää ei selainta suljettaessa kirjata sovelluksesta ulos, käytetään julkisella tietokoneella, on tietokoneen seuraavalla käyttäjällä pääsy edellisen käyttäjän istuntoon. [OWASP, 2013]

Turvallinen todentaminen on toteutettu niin, että hyökkääjä ei pysty lukemaan käyttäjän tunnuksia, vaikka pääsisikin käsiksi palvelimelle lähetettäviin kutsuihin. Pyynnöt ja vastaukset kannattaa esimerkiksi salata käyttämällä HTTPS-yhteyttä. Istunto kannattaa myös asettaa vanhenemaan mahdollisimman pienen käyttämättömän ajan kuluttua, jotta istuntoa ei voi hyödyntää esimerkiksi julkisilla tietokoneilla kirjautunutta käyttäjää seuraava käyttäjä. [OWASP, 2013]

2.4. Objektien auktorisointi

Web-sovellusten turvallisuus perustuu sekä käyttäjän todentamiseen että auktorisointiin. Koska web-sovellukset hakevat informaation URL:n perusteella, saattaa heikosti tai epäjohdonmukaisesti toteutettu auktorisointi mahdollistaa pääsyn sellaisiin objekteihin, joi-

hin käyttäjällä ei pitäisi olla pääsyä, pelkän URL-manipulaation avulla [OWASP, 2013]. Sovellus voisi esimerkiksi tarjota blogikirjoituksia käyttäjän määrittämälle käyttäjäjoukolle. Kirjoitukset tallennetaan SQL-tietokantaan ja yksittäiseen kirjoitukseen viitataan suoraan URL:n lopussa kirjoituksen järjestysnumerosta johdetun id:n avulla. Jos sovelluksen turvallisuus nojaa oletukseen, että käyttäjälle ei tarjota linkkiä sellaiseen blogikirjoitukseen, johon käyttäjällä ei pitäisi olla pääsyä, voi vihamielinen käyttäjä päästä käsiksi mihin tahansa blogikirjoitukseen pelkästään muokkaamalla URL:ssa olevaa tunnistetta.

Viittausten aiheuttamaan riskiin voidaan varautua kahdella tavalla. Sovellus voidaan toteuttaa esimerkiksi niin, että suoria viittauksia objekteihin ei ole, vaan että jokainen viittaus on istuntokohtainen ja ne tulkitaan viittamaan alkuperäiseen objektiin sovelluksen palvelinpuolella. Toinen vaihtoehto on sallia suorat viittaukset ja varmistaa järjestelmällisesti jokaisessa kutsussa, että kyseisellä käyttäjällä on pääsy haettuun objektiin. [OWASP, 2013]

2.5. Funktioiden auktorisointi

Funktioiden auktorisoinnilla varmistetaan, että sovelluksen tarjoamia funktioita pääsee hyödyntämään vain ne käyttäjät, joille pääsy on tarkoitettu. Auktorisoinniksi ei funktioidenkaan osalta riitä pelkkä linkkien piilottaminen, vaan vihamielinen käyttäjä voi yrittää pääsyä rajattuihin funktioihin URL-manipulaation avulla. [OWASP, 2013]

Funktioiden auktorisointi tulee hajauttaa erilliselle komponentille, jonka toiminta tulee olla helposti varmistettavissa. Oletuksena kaikkien funktioiden tulee olla rajattuna kaikilta käyttäjiltä, ja rajausta tulee höllentää vain tarvittaessa halutulle käyttäjälle tai käyttäjäryhmälle. [OWASP, 2013]

2.6. Virheelliset turvallisuuskonfiguraatiot

Web-sovellusten toteuttamiseen tarkoitettuja ohjelmointikieliä, tietokantoja ja ohjelmistokehyksiä (framework) on lukuisia, ja myös turvallisuus on toteutettu eri tekniikoissa eri tavalla. Osa ohjelmistokehyksistä pakottaa kehittäjän asettamaan tärkeimmät turvallisuusasetukset itse ennen kuin koko ohjelmistokehystä pääsee ylipäätään työstämään, kun taas toiset ovat täysin käyttövalmiita ZIP-paketin purkamisen ja tietokanta-asetusten sisällyttämisen jälkeen. Sovelluskehittäjän vastuulle jää aina turvallisuusasetusten varmistaminen. Hyvin dokumentoitu sovelluskehys tarjoaa kattavan ohjeistuksen turvallisuusasetusten läpikäyntiin ja toimivuuden varmistamiseen.

Jos kehittäjää ei ohjeisteta heti sovelluskehystä asentaessa säätämään turvallisuusasetuksia kuntoon, on mahdollista, että sovellukseen asetetut oletusasetukset jäävät voimaan. Jos sekä kehyksen järjestelmänvalvojalle tarkoitetun pääkäyttäjäpaneelin URL että käyttäjätunnukset jätetään oletusasetusten mukaisiksi, saattaa koko sovelluksen kaappaaminen, tietokantaan tallennettujen tietojen varastaminen ja tyhjentäminen sekä sivuston sisällön

muokkaaminen olla erityisen helppoa kehystä aiemmin käyttäneelle hyökkääjälle. [OWASP, 2013]

2.7. Arkaluonteisen tiedon paljastaminen

Jotta web-sovellusten vuorovaikutus on mahdollisimman helppoa ja yksinkertaista, on kriittisenkin tiedon tallentaminen tietokantaan ja välimuistiin tavallista. Sovellusten tietokantoihin ja käyttäjän selaimeen joudutaan tallentamaan turvallisuuden kannalta kriittistä tietoa, kuten salasanoja, henkilötunnuksia, luottokorttitietoja, sairaskertomuksia, reseptejä ja niin edelleen, joten datan suojaaminen tiivistämällä tai salaamalla on tärkeä osa turvallisen sovelluksen toimintaa. [OWASP, 2013]

Tallennettuun tietoon voidaan päästä käsiksi lukemalla tieto suoraan tietokannasta tai niin sanotun mies välissä -hyökkäyksen (man-in-the-middle attack) avulla, eli lukemaan tieto palvelimen ja sitä hyödyntävän asiakkaan välisistä transaktioista. Datan suojaaminen pelkästään tietokantaan syötettäessä ei siis riitä, vaan myös transaktioiden sisältö on tärkeä salata esimerkiksi SSL/TLS-salauksen eli käytännössä HTTPS-yhteyden avulla. [OWASP, 2013]

2.8. Cross-site request forgery

Cross-site request forgery, lyhemmin CSRF, on hyökkäys, jossa käyttäjän ladattavaksi tarkoitettuun sisältöön upotetaan ohjelmakoodia, jota suorittaessaan selain tekee pyynnön toiselle sivustolle. Jos käyttäjä on samaan aikaan kirjautuneena ohjelmakoodin osoittamaan sivustoon, haitallinen suoritus tehdään käyttäjän tietämättä käyttäjän istunnon valtuutuksella. CSRF:ään johtava ohjelmakoodi on normaalisti osa sivuston dynaamista sisältöä, kuten käyttäjien lataamia kuvia, tekstiä tai esimerkiksi mainosbannereita. Esimerkiksi viattoman näköisen mainoksen suorittaminen selaimessa voikin käyttäjän tietämättä suorittaa ostoskoritapahtuman käyttäjän tunnuksilla toisella sivustolla. [OWASP, 2013]

CSRF:ään tulee varautua sekä sivustolla, josta ohjaus tehdään, että sivustolla, johon hyökkäys kohdistuu. Jotta hyökkäys ei onnistuisi, tulee palvelimelle lähetettävien tietojen mukaan sisällyttää niin sanottu CSRF-avain, jonka oikeellisuus varmistetaan ennen pyynnön suorittamista. CSRF-avain luodaan palvelimella, ja se voidaan esimerkiksi upottaa lomakkeen piilotettuun kenttään. Turvattomampi vaihtoehto on siirtää avain GET-parametrina, jolloin avain on ulkopuolisen luettavissa. Olennainen osa CSRF-avaimen toimintaa on mahdollisimman pieni voimassaoloaika, jotta vääriin käsiin joutunut avain ei mahdollista hyökkäystä myöhemmin. [OWASP, 2013]

2.9. Haavoittuneiden komponenttien käyttäminen

Koska mitä tahansa sovellusta toteuttaessa yksi tehokkaan ja tietoturvallisen kehittämisen perusta on olemassa olevien ratkaisuiden käyttäminen, eli käytännössä ulkoisten komponenttien ja lisäosien hyödyntäminen, tulee myös niiden turvallisuuteen kiinnittää huomio-

ta. Avoin lähdekoodi mahdollistaa komponenttien turvallisuuden varmistamisen kooditasolla, mutta myös uhkien suunnittelemisen kehittäjien tietämättä. On myös mahdollista, että sovellukseen on jätetty tahallisia takaportteja komponentin kehittäjien omien intressien mukaisesti. [OWASP, 2013]

Koska sovelluksen rakentaminen ilman ulkoisia komponentteja on harvoin mahdollista, on tärkeä käyttää komponentteja, joita yhä kehitetään sekä varmistaa, jotta käytössä on aina mahdollisimman tuore ja vakaa versio käytetystä komponentista. Usein yksittäinen komponentti saattaa olla riippuvainen muista komponenteista, joten myös niiden turvallisuus tulee arvioida. Avoimeen lähdekoodiin kuuluu normaalisti myös komponenttien ja lisäosien julkinen kommentointi ja arviointi, minkä seuraaminen helpottaa turvallisuuden varmistamista. Hyödynnettävistä komponenteista voi olla myös mahdollista eristää omaan käyttöön vain ne osat, jotka kehittäjä kokee tarpeelliseksi ja jättää ylimääräiset toiminnollisuudet toteutettavan järjestelmän ulkopuolelle. [OWASP, 2013]

2.10. Validoimattomat linkit ja uudelleenohjaukset

Verkkosovellukset, joihin käyttäjät voivat syöttää URL-osoitteita muiden käyttäjien ohjaamiseen, saattavat olla riskialttiita validoimattomille uudelleenohjauksille. Hyökkääjä voi esimerkiksi manipuloida ohjaamista niin, että käyttäjä ohjataan sivustolle, joka suorittaa käyttäjän tietokoneelle haitallista ohjelmakoodia ja esimerkiksi lukee tai poistaa tietoja käyttäjän kovalevyltä. Taitavasti toteutettu ohjaaminen voi tapahtua niin, että käyttäjä ei edes huomaa uudelleenohjausta sivuston ulkopuolelle tapahtuneeksi, minkä takia haittaohjelman suoritus vaikuttaisi olevan alkuperäisen sivuston aikaansaamaa. Validoimattomat uudelleenohjaukset ovat yksi suositummista tavoista kalastella käyttäjätietoja alkuperäisen sivuston luottamukseen nojautuen [OWASP, 2013]

Yksinkertaisin tapa validoimattomien uudelleenohjausten estämiseksi on estää käyttäjiä luomasta kustomoituja uudelleenohjauksia. Käyttäjä ei siis saa antaa millekään toisen käyttäjän tietokoneella suoritettavalle uudelleenohjaukselle parametreja, joilla uudelleenohjaaminen sivuston ulkopuolelta haitalliselle sivustolle olisi mahdollista. Kaikkien sivuston uudelleenohjausten tulee siis olla staattisesti määriteltyjä. Käyttäjän syötteet tulee muutoinkin validoida niin, että käyttäjä ei voi aiheuttaa uudelleenohjausta esimerkiksi cross site scripting -hyökkäyksen avulla. [OWASP, 2013]

Jos sovelluksen toiminnan kannalta on välttämätöntä mahdollistaa käyttäjien kustomoimat uudelleenohjaukset, tulee jokainen ohjaus auktorisoida ja validoida. Tärkeintä on varmistaa, että käyttäjä ei pysty luomaan sivustolle kokonaista URL:ia, vaan että muun muassa kyselyparametrit lisätään URL:iin erikseen. [OWASP, 2013]

3. Todentaminen

Web-palveluissa ja tietojärjestelmissä ylipäätään yksi olennaisimmista ongelmista on varmistaa, että pyyntöjä tekevä käyttäjä on se, joka hän väittää olevansa. Perinteinen tunnistaminen esimerkiksi passin tai henkilökortin avulla ei ole tietotekniikassa käytännöllinen tapa, joten käyttäjän tunnistamista varten on kehitetty useita muita vaihtoehtoisia keinoja. Todentaminen tai autentikointi on tietoturvallisuuteen läheisesti liittyvä tehtävä, jossa tarkoituksena on varmistaa käyttäjän identiteetti.

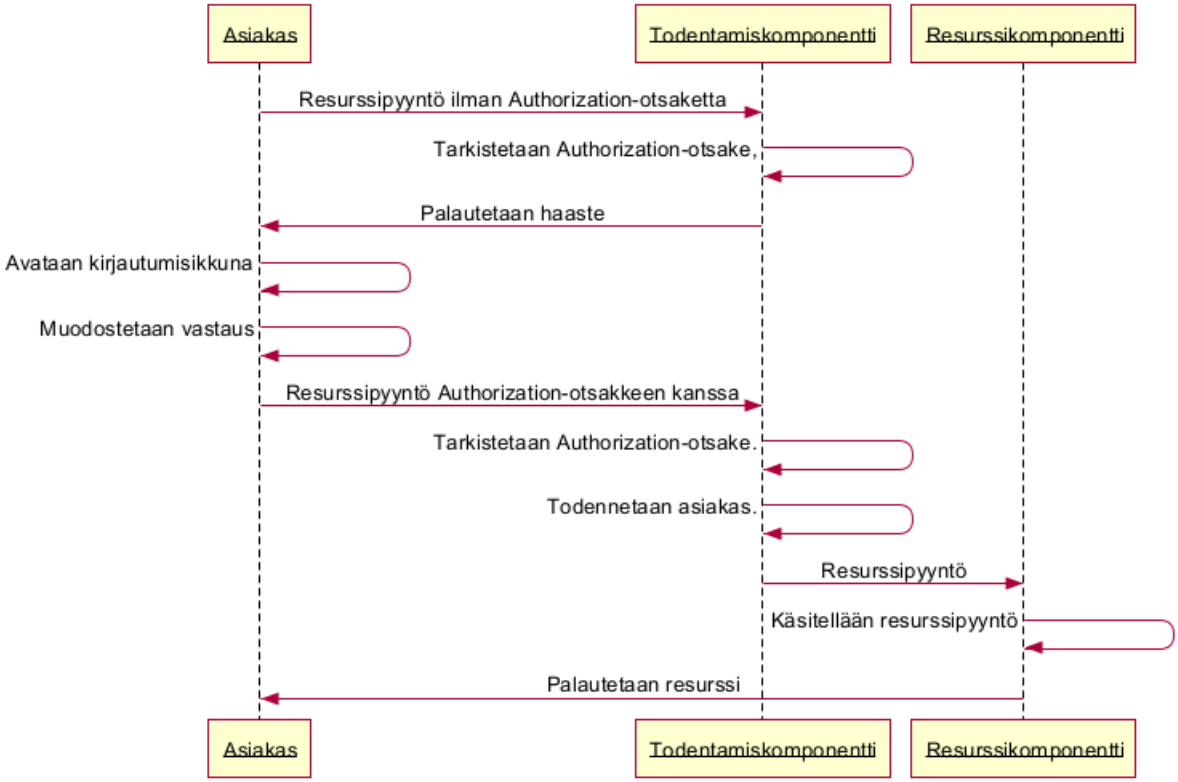
Sovellusten monimuotoisuus on johtanut myös todentamistapojen monimuotoisuuteen. Käyttäjän oikean identiteetin, kuten nimen, osoitteen ja varsinkaan henkilötunnuksen varmistamiselle ei läheskään aina ole tarvetta, eikä käyttäjä toisaalta halua välttämättä edes antaa palvelun tuottajalle tietoa oikeasta identiteetistään. Kriittisimpiä palveluita tuottaessa taas esimerkiksi perinteinen henkilökorttiin perustuva tunnistaminen voi olla aivan liian yksinkertainen tapa varmistaa käyttäjän henkilöllisyys.

Turvallisuuden kannalta vähemmän kriittisissä järjestelmissä ja sovelluksissa todentaminen perustuu johonkin, mitä käyttäjän tietää, kuten käyttäjätunnuksen ja salasanan yhdistelmään, jolloin käyttäjä lähettää palvelimelle oman yhdistelmänsä ja saa tunnuksiaan vastaan pääsyn haluamiinsa resursseihin tai funktioihin. Kriittisemmissä sovelluksissa käyttäjän vaaditaan usein pelkän tunnusten tietämisen lisäksi pitävän hallussaan jotain yksilöllistä esinettä tai ominaisuutta, jonka avulla voidaan varmemmin vahvistaa käyttäjän olevan se, joka hän väittää olevansa.

HTTP-standardissa määritettyjen rajoitusten mukaisesti palvelin on tilaton, joten kaikki edelliseen pyyntöön liittyneet muuttujien, taulukoiden tai olioiden asetukset eivät enää toisen pyynnön yhteydessä ole voimassa. Eri pyyntöjen välillä hyödynnettävä tieto, kuten kirjautuneen käyttäjän todentamiseen tarvittava informaatio, tulee siis erikseen tallentaa joko tietokantaan, evästeisiin, palvelimen keskusmuistiin tai selaimen välimuistiin, jotta käyttäjän ei tarvitsisi itse autentikoitua jokaisen pyynnön yhteydessä, vaan että järjestelmä pitää itse huolta, että tieto kirjautuneesta käyttäjästä kulkee palvelimelle jokaisen pyynnön yhteydessä käyttäjän siihen puuttumatta. Tässä tutkielmassa esitellään kahdeksan toisistaan poikkeavaa todentamistekniikkaa ja -standardia sekä analysoidaan niitä sekä kehittämisen tehokkuuden että tietoturvan kannalta.

3.1. HTTP-todentaminen

HTTP-todentaminen on perinteinen todentamisen keino, joka perustuu käyttäjätunnuksen ja salasanan yhdistelmään käyttäjän identiteetin todentamiseksi [Berners-Lee et al., 1996]. HTTP-todentamisessa selaimen vastuulle jää käyttäjän tunnusten lähettäminen palvelimelle jokaisen pyynnön yhteydessä joko salaamattomana tai salattuna. Kuvassa 1 on esitetty HTTP-todentaminen sekvenssikaavion avulla.



Kuva 1: Sekvenssikaavio HTTP-todentamisesta

HTTP-todentaminen perustuu haasteeseen ja vastaukseen (challenge and response). Haasteessa kerrotaan asiakkaalle, minkälaisesta todentamisesta on kyse sekä annetaan tarvittaessa lisätietoa todentamiseen liittyen, kuten salausalgoritmi, jolla tunnukset salataan. Kaksi tunnetuinta HTTP-todentamisen toteutusta ovat HTTP Basic access -todentaminen sekä Digest access -todentaminen.

3.1.1. HTTP Basic access -todentaminen

Basic access on jo vuonna 1996 HTTP-standardin versiossa 1.0 esitetty yksinkertainen ja jo julkaistessa turvattomaksi todettu todentamistoteutus, joka ei toimiakseen vaadi istuntoa tai evästeitä [Berners-Lee et al., 1996]. Jos palvelimelta kysytty resurssi vaatii tunnistamista, palvelin vastaa kyselyyn haasteella, johon asiakkaan on vastattava, jotta tunnistaminen onnistuu. Basic access -todentamisessa kirjautuminen tapahtuu selaimen oman kirjautumisikkunan avulla ja käyttäjätunnus ja salasana lähetetään palvelimelle Base64-koodattuna HTTP-pyynnön Authorization-otsakkeessa. [Chapman and Chapman, 2012]

Basic access -todentaminen voidaan kuvata seuraavasti [Chapman and Chapman, 2012]:

1. Asiakas lähettää palvelimelle pyynnön
2. Palvelin tarkistaa pyynnön Authorization-otsakkeen, ja jos otsakkeessa ei ole kirjautumiseen vaadittuja tunnuksia, palauttaa se asiakkaalle haasteen eli statuksen 401

sekä esimerkiksi viestin WWW-Authenticate: Basic realm=”Auhorized personnel only”

3. Selain tulkitsee statuksen 401 perusteella palvelimen vaativan todentamista ja avainsanasta Basic palvelimen vaatimaan nimenomaan Basic access -todentamista. Selain avaa käyttäjälle todentamisikkunan, johon käyttäjä voi kirjoittaa käyttäjätunnuksen sekä salasanan ja yrittää kirjautumista.
4. Selain tekee saman pyynnön uudelleen, mutta lisää kyselyn Authorization-otsakkeeseen sanan Basic sekä Base64-koodattuna ja kaksoispisteellä toisistaan eroteltuina käyttäjätunnuksen ja salasanan. Jos käyttäjä syöttäisi kirjautumisikkunaan käyttäjätunnuksen esimerkkitunnus ja salasanan esimerkkisalasana, olisi pyynnön mukana lähetettävä otsake seuraavanlainen: Authorization: Basic ZXNpbWVya2tpdHVubnVzOmVzaW1lcmtraXNhbGFzYW5h.
5. Jos palvelin ei vastaa pyyntöön virheviestillä, lisää selain edellä mainitun otsakkeen kaikkiin palvelimelle lähetettäviin pyyntöihin jatkossa.

Basic access -todentamista toteuttaessa tulee ottaa huomioon, että Base64 ei ole keino salata merkkijonoa, vaan pelkästään varmistaa, että alkuperäisten merkkien sisältämä informaatio ei katoa merkistöristiriidoista johtuen. Salaamattomuudesta johtuen käyttäjätunnus ja salasana ovat helposti koodattavissa takaisin alkuperäiseen muotoonsa, joten ilman HTTPS-yhteyttä Basic access -todentaminen on erityisen altis mies-välissä -hyökkäyksille. [Chapman and Chapman, 2012]

Tietoturvan kannalta erityisen kriittistä on myös se, että palvelin ei pysty vaikuttamaan mitenkään käyttäjän uloskirjaamiseen. Välimuisti, johon käyttäjätunnus ja salasana ovat selaimessa tallennettuina, on voimassa juuri niin kauan kuin tunnuksiin tehdään muutoksia palvelimella. Basic access -todentamisen yhteydessä ei voida määritellä myöskään minkäänlaista istunnon voimassaoloaikaa eli niin kauan kuin selaimen tallentamaa otsaketta ei tyhjennetä, voi kuka tahansa hyödyntää käyttäjän kirjautumista. [Chapman and Chapman, 2012]

Basic access -todentamisen ongelmana voidaan myös pitää selaimen automaattisesti tarjoamaa kirjautumisikkunaa, jonka ulkonäköön kehittäjä ei pysty vaikuttamaan. Toisaalta on myös mahdollista, että selain itsessään ei tue Basic access -todentamista. Tällöin sovellus, joka pyrkii tukemaan mahdollisimman montaa erilaista selainta, joutuu tarjoamaan kirjautumista varten myös sivustolle upotetun lomakkeen kirjautumistietojen syöttämistä varten.

3.1.2. HTTP Digest access -todentaminen

Myös Digest access -todentaminen perustuu palvelimen lähettämään haasteeseen ja asiakkaan vastaukseen, mutta toisin kuin Basic access -todentamisessa, jossa käyttäjätunnus ja salasana vain koodataan Base64-algoritmilla, digest access -todentamisessa tunnukset salataan MD5-algoritmin avulla. Digest access -todentamista voidaan pitää turvallisena vaihtoehtona erityisesti silloin, kun liikennettä ei voida salata TLS/SSL-salauksella, vaan käytössä on suojaamaton HTTP-yhteys.

Digest access -todentaminen tapahtuu seuraavasti [Chapman and Chapman, 2012]:

1. Asiakas lähettää palvelimelle pyynnön.
2. Pavelin toteaa pyydetyn resurssin vaativan todentamista ja palauttaa asiakkaalle haasteena statuksen 401 ja esimerkiksi viestin WWW-Authenticate: Digest realm=”Securely authorized personnel only” nonce= “dcd98b7102dd2f0e8b11d0f600bfb0c093” qop=”auth”.
3. Asiakas tunnistaa palvelimen vaativan Digest access -todentamista 401-statuksen ja Digest-avainsanan perusteella ja avaa käyttäjälle kirjautumisikkunan.
4. Selain tekee uuden pyynnön seuraavanlaisella otsakkeella: Authorization: Digest username=”esimerkkitunnus”, realm=”Securely authorized personnel only”, nonce=”dcd98b7102dd2f0e8b11d0f600bfb0c093”, uri=”/resurssi”, algorighm=MD5, response=” 6629fae49393a05397450978507c4ef1”, qop=”auth”.
5. Palvelin tulkitsee arvot, eli varmistaa, että nonce on sama, jonka palvelin oli kyseiselle asiakkaalle luovuttanut, käyttäjätunnus on olemassa ja että salasana vastaa kyseistä käyttäjätunnusta sekä usein myös varmistaa, että nonceen sisällytetty aika-arvo ei ylitä istunnolle määriteltyä vanhenemisaikaa.

Palvelimen palauttaman haasteen merkittävin arvo on nonce (lyhenne sanoista number used once). Nonce on yksilöllinen merkkijono, joka sisällytetään myös selaimen lähettämään vastaukseen. Noncen avulla vastaus voidaan tunnistaa oikean asiakkaan lähettämäksi, ja koska nonce sisältää normaalisti myös aika-arvon, voidaan sen perusteella myös päättää, onko käyttäjän istunto vanhentunut. [Chapman and Chapman, 2012]

Selaimen haasteeseen vastaava vastaus on merkittävästi monimutkaisempi kuin Basic access -todentamisessa. Noncen ohella tärkeä osa vastausta on response, jossa itse MD5-salaus on hyödynnetty. Response koostuu MD5-salatuista arvoista HA1, nonce, nc, cnonce, qop ja HA2. HA1 on MD5-salattu yhdistelmä pilkulla erotetuista käyttäjätunnuksesta, realm-arvosta ja salasanasta. HA2 on MD5-salattu yhdistelmä kyselyn metodista(GET, POST, DELETE, PUT tai PATCH) sekä URL:sta. [Chapman and Chapman, 2012]

Digest-todentaminen on Basic access -todentamista merkittävästi turvallisempi ratkaisu, ja vaikka MD5-salaus on periaatteessa purettavissa, Digest-autentikaatiota voidaan pitää turvallisena myös salaamattomalla HTTP-yhteydellä käytettynä [Chapman and

Chapman, 2012]. Digest-autentikaatiossa on kuitenkin ongelmansa, sillä se on standardina jo vanha ja tarkoitettu perinteisellä web-selaimella käytettäväksi. Jos Digest-haasteen saakin selain, joka ei tue Digest-autentikaatiota, jää monimutkaisen vastauksen muodostaminen asiakassovellusta toteuttavan kehittäjän vastuulle. Digest-todentamisessa käytetty selaimen oma kirjautumisruutua taas voidaan pitää esteettisesti epämiellyttävänä, eikä kehittäjä pysty mitenkään vaikuttamaan sen ulkonäköön [Chapman and Chapman, 2012]. Jos digest-autentikaatiota tukeva selain saa digest-tyyppisen haasteen, ei ole varmaa, voidaanko vanhanaikaisen kirjautumisikkunan näyttämistä estää kaikilla selaimilla.

3.2. Istuntoihin perustuva todentaminen

Istuntoihin perustuvalla todentamisella (session based authentication) viitataan tekniikkaan, jossa valideja tunnuksia vastaan asiakkaan ja palvelimen välille luodaan istunto, joka pitää sisällään tiedon kirjautuneesta käyttäjästä. Istunto voidaan tallentaa joko käyttäjän selaimen evästeisiin, palvelimen tietokantaan tai palvelimen keskusmuistiin.

Kun istunto tallennetaan evästeisiin, istuntoihin sisällytetty data kulkee palvelimelle jokaisen pyynnön yhteydessä selaimen lähettämänä automaattisesti. Koska kuka tahansa kyseiselle tietokoneelle päässyt henkilö voi lukea ja muokata evästeitä, on istuntoihin tallennettava tieto tällöin tietoturvasyistä kannattavaa salata sekä välttää tallentamasta istuntoihin mitään arkaluontoista tietoa [Chapman and Chapman, 2012]. Evästeisiin tallennettu istunto lopetetaan tyhjentämällä evästeen sisältö, mikä ei kutenkaan tarkoita, etteikö evästettä voisi kopioida ja käyttää siten vielä uloskirjaamisen jälkeen uudelleen.

Koska eväste kulkee palvelimelle jokaisen kutsun yhteydessä, se kuormittaa verkkoyhteyttä. Toisaalta asiakassovelluksen ylläpitämä tieto kirjautuneesta käyttäjästä helpottaa palvelimen kuormaa ja näin edesauttaa tehokkaan palvelinpään tarjoamista pienemmillä resursseilla. Evästeiden rajoituksena voidaan myös pitää istuntoihin tallennettavan tiedon kokonaismäärää, joka on rajattu useissa vanhemmissa selaimissa 4 kilobittiin [Chapman and Chapman, 2012].

Jos istuntoihin halutaan tallentaa arkaluontoista dataa kuten luottokortti- tai henkilötietoja tai jos tallennettavan tiedon koko saattaa ylittää neljän kilobitin rajan, voidaan istunto tallentaa myös palvelimen keskusmuistiin tai tietokantaan. Palvelimen puolelle tallennettu istunto kuormittaa palvelinta ja vaatii varsinkin suuria asiakasmääriä varten merkittävää panostusta sekä prosessointitehoon että kiintolevy- tai keskusmuistikapasiteettiin riippuen evästeiden tallennuspaikasta. Jotta palvelimelle tallennettu istunto voidaan yhdistää yksittäiseen käyttäjään, istunnon tunniste tallennetaan onnistuneen kirjautumisen jälkeen evästeeseen, jonka asiakas palauttaa palvelimelle kunkin pyynnön yhteydessä. [Chapman and Chapman, 2012]

Istunnon tunnistetta luotaessa tulee huomioida erityisesti mies välissä -hyökkäys, jottei tunnisteesta itsestään voi päätellä, kuinka sen muodostaminen tapahtuu. Näin voidaan varmistaa, ettei mahdollinen hyökkääjä voi generoida muuhun käyttäjään liitettyä tunnis-

tetta ja kaapata tämän istuntoa. Tunnisteelle voidaan määrittää myös voimassaoloaika, jolla pyritään minimoimaan istunnon hyväksikäytön mahdollisuus. [Chapman and Chapman, 2012]

3.3. Suojaustunnukseen perustuva todentaminen

Suojaustunnuksiin eli tokeneihin perustuvassa todentamisessa käyttäjän kirjautumistunnukset vaihdetaan kirjautumisen yhteydessä suojaustunnukseen, jonka selain tai muu asiakassovellus tallentaa välimuistiinsa ja lähettää muiden kutsujen yhteydessä käyttäjän identiteetin todisteena [Cheng et al., 2015]. Suojaustunnus voidaan palvelimen puolella salata ja allekirjoittaa, jolloin palvelin voi seuraavien kutsujen yhteydessä tarkistaa tunnuksen oikeellisuuden. Suojaustunnukseen voidaan myös tallentaa tietoa, kuten aikaleimoja tai kokonaisia objekteja avain—arvo-pareina.

Suojaustunnuksen tärkein ominaisuus on autentikaation kannalta se, että palvelimen itsensä ei tarvitse lainkaan tallentaa tunnusta, vaan se voi suoraan tunnuksesta päätellä, mitä tietokantaan tallennettua käyttäjää tunnus koskee ja onko tunnus validi. Suojaustunnus siis käytännössä vastaa istuntoa, mutta toisin kuin istunnoissa, suojaustunnus ei vaadi istuntotietojen tallentamista palvelimen omaan muistiin tai tietokantaan. Suojaustunnuksen sisältö voi myös olla salauksen takia ulkopuolisten saavuttamattomissa ja siksi suojaustunnusta voidaan pitää jossain määrin turvallisena myös salaamattoman HTTP-yhteyden läpi käytettynä, varsinkin, jos tunnus on kertakäyttöinen, eli tunnus vaihdetaan uuteen aina jokaisen palvelinkutsun yhteydessä [Cheng et al., 2015]. Asiakassovellus voi lähettää suojaustunnuksen palvelimelle esimerkiksi evästeiden tai otsakkeiden osana.

Myös suojaustunnuksen huolettomassa käyttämisessä on riskinsä. Jos suojaustunnusta ei ole asetettu vanhenemaan, vakavin uhka on mies välissä -hyökkäys ja sen avulla toteutettu istunnon kaappaaminen, jolloin hyökkääjä voi esiintyä hyökkäyksen uhrina ja näin hallita uhrin käyttäjätiliä. Erityisen riskialtista suojatunnuksen käyttäminen on, jos tunnukseen ei ole määritelty käyttöikää, sillä siinä tapauksessa hyökkääjä voi käyttää tunnusta, kunnes tunnus saadaan jollain keinotekoisella tavalla vanhenemaan.

Koska suojaustunnukseen voidaan tallentaa mitä tahansa esimerkiksi käyttäjään liittyvää tietoa, voi tunnuksen tietosisällön joutuminen vääriin käsiin olla haitallista myös muutoin kuin istunnon kaappaamisen kannalta. Suojaustunnuksen muodostamiseen tulee myös kiinnittää huomiota, jotta tunnuksesta itsestään ei voi päätellä, kuinka sen muodostaminen tapahtuu. Jos muodosta voidaan päätellä, kuinka validin tunnuksen luominen onnistuu, voi tiedon avulla pahimmassa tapauksessa kaapata istunnon ilman mies välissä -hyökkäystä.

Suojaustunnus myös vie tilaa ja koska selain lähettää sen jokaisen palvelimelle tehtävän pyynnön yhteydessä, suurikokoisella suojaustunnuksella voidaan myös varata kohtuuttomasti kaistaa palvelimen ja asiakkaiden välisestä verkkoyhteydestä. Suojaustunnus-

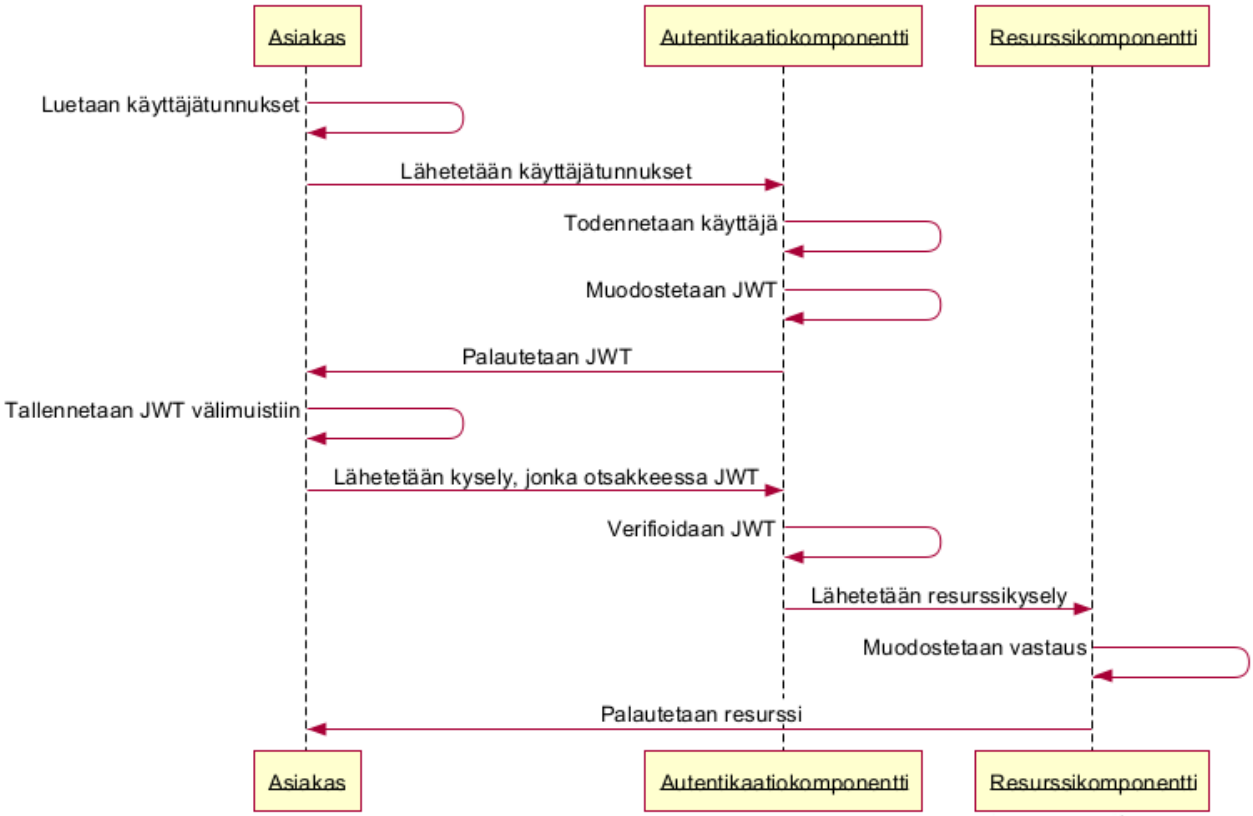
tyyppiä valitessa kannattaa siis käyttää sellaista formaattia, joka vie mahdollisimman vähän tilaa ja itse tunnusta muodostaessa tulee välttää ylimääräisen tiedon tallentamista tunnuksen tietosisältöön.

Tunnetuimpia suojaustunnustoteutuksia ovat Security Assertion Markup Language eli SAML ja JSON Web Token eli JWT sekä Simple Web Token eli SWT. Tässä tutkielmassa tutustaan lähemmin vain JSON Web Tokenin avulla toteutettuun autentikaatioon, sillä SAML:ssa tunnukseen pakataan merkittävästi enemmän tilaa vaativaa XML:ää ja SWT taas on salauksensa puolesta vaatimattomampi kuin JWT tai SAML. [JWT, 2015]

JSON Web Token, eli JWT, on standardi, jonka mukaisesti JSON-objekti (JavaScript Object Notation Object) voidaan tiiviistää merkkijonoksi JSON Web Encryptionin (JWE) ja/tai JSON Web Signaturen (JWS) avulla [Bradley et al., 2014]. JWT koostuu otsakkeesta (header), tietosisällöstä (payload) ja allekirjoituksesta (signature). Kukin osa on JSON-muodossa, mutta JSON:n syntaksista poiketen osat erotellaan toisistaan pisteiden avulla. JWT:n otsake sisältää metatietoa JWT:stä, kuten tyypin (JWT:n tapauksessa tyyppi on JWT) ja salausalgoritmin kuten HMAC SHA256 tai RSA. Tietosisällössä on itse JWT:hen pakattu data. JWT:n Allekirjoitus muodostetaan seuraavasti [Bradley et al., 2014]:

1. Koodataan Base64-koodauksen avulla JWT:hen pakattava otsake sekä tietosisältö.
2. Yhdistetään edellä koodatut merkkijonot toisiinsa pisteen avulla.
3. Muodostetaan salaisuus (secret).
4. Salataan merkkijono esimerkiksi HMAC SHA256 -salauksen avulla käyttäen salausavaimena kohdassa 3. muodostettua salaisuutta.

JWT-standardi ei pakota mitään avainpareja käytettäväksi, mutta muiden muassa seuraavat Bradley ja muut [2014] luettelevat todennäköisesti erilaisten JWT:n käyttötapojen kannalta hyödyllisiksi: iss (issuer), sub (subject), aud (audience), exp (expiration time), nbf (no before), iat (issued at) sekä jti (JWT id). Autentikaation tapauksessa olennaisimpia avaimia ovat sub, exp sekä iat, joista sub pitää sisällään tiedon sovelluksen tietokantaan tallennetun käyttäjän id:stä, exp JWT:n viimeisen voimassaloajan ja iat JWT:n luomisajan.



Kuva 2: Sekvenssikaavio JWT-perustaisesta todentamisesta

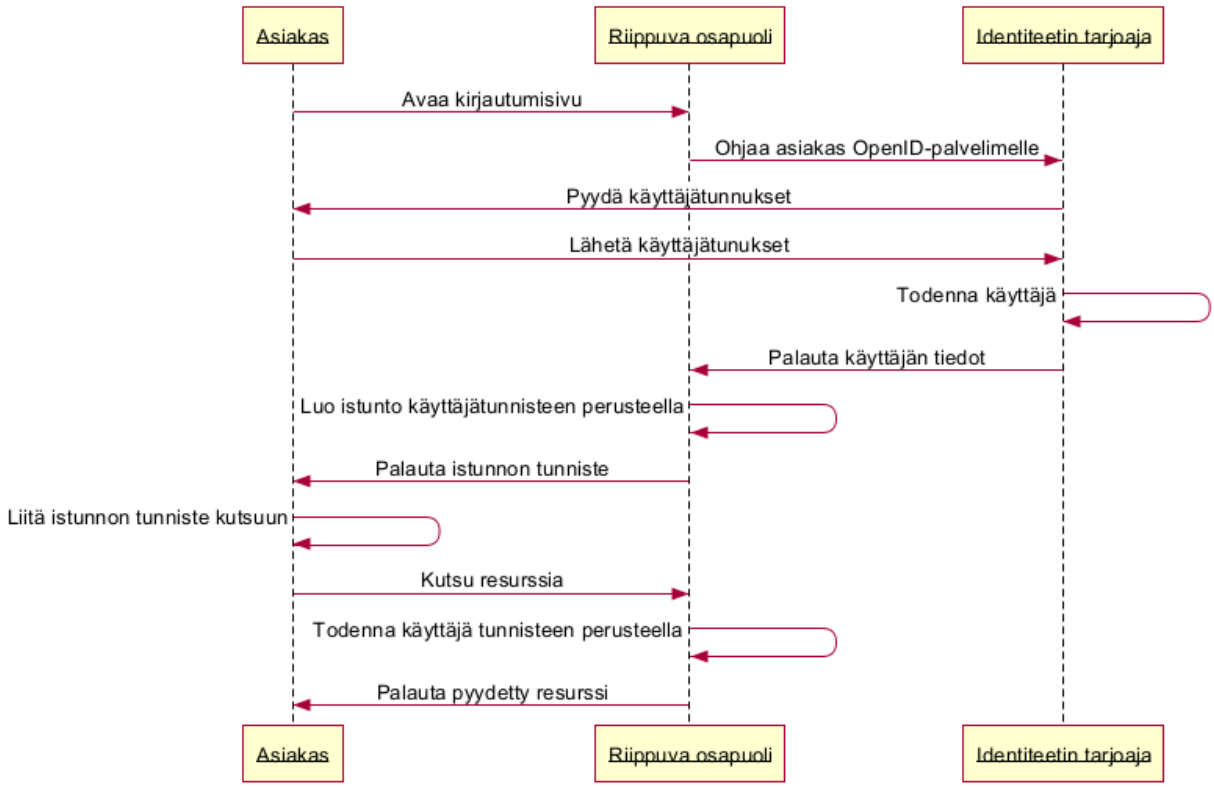
Kuvan 2 sekvenssikaaviossa on esimerkkitapaus JWT-perustaisesta autentikaatiosta. Autentikaatiosekvensessi alkaa, kun käyttäjä syöttää käyttäjätunnuksensa ja salasanansa lomakkeelle, josta tunnukset lähetetään palvelimelle. Jotta tunnukset eivät olisi alttiita mies välissä -hyökkäykselle, on tärkeää käyttää tunnuksien lähettämiseen salattua HTTPS-yhteyttä. Kun tunnukset saapuvat palvelimelle, autentikaatiokomponentti varmistaa, että järjestelmän tietokannasta löytyy käyttäjä, johon annetut tunnukset täsmäävät.

Kun tunnuksia vastaava käyttäjä on löytynyt, muodostetaan JWT luomisajankohdasta (iss), mieluiten mahdollisimman lyhyestä voimassaoloajasta (exp) sekä tietokannasta löytyneen käyttäjän id:stä (sub). Aikaleimoissa käytetään UNIX-aikaa. Koska JWT:n standardi on tietosisällön suhteen avoin, voi JWT:hen tiivistää tarpeen tullen myös muuta tietoa, kuten käyttäjätietoja, joita halutaan kirjautuneesta käyttäjästä hyödyntää sovelluksen selainpuolella. JWT:hen tiivistetty tieto kannattaa kuitenkin minimoida, jotta palvelimelle lähettävät kyselyt voidaan pitää mahdollisimman pieninä.

3.4. OpenID

OpenID on todentamistapa, jossa käyttäjätilin hallinnointi on ulkoistettu erilliselle todennuspalveluntarjoajalle. Käyttäjän tarvitsee siis rekisteröityä vain yhteen palveluun ja hyö-

dyntää todennuspalvelun ylläpitämää identiteettiä muilla sivustoilla, jotka tukevat OpenID:tä.



Kuva 3: Sekvenssikaavio OpenID-todentamisesta

Kuvassa 3 esitelty OpenID-todentaminen koostuu identiteetin tarjoajasta (identity provider), riippuvasta osapuolesta (relying part) sekä käyttäjästä (consumer). Riippuva osapuoli tarjoaa käyttäjälle mahdollisuuden kirjautua OpenID:tä hyödyntäen, jolloin käyttäjä ohjataan identiteetin tarjoajan sivustolle, jossa käyttäjä tunnistautuu esimerkiksi käyttäjätunnuksen ja salasanan avulla. Kun käyttäjä on onnistuneesti tunnistautunut, varmistetaan käyttäjältä, että kirjautuminen riippuvan osapuolen sivustolle hyväksytään. Hyväksymisen jälkeen käyttäjä ohjataan takaisin sivustolle, johon hän alun perin halusi kirjautua. Ohjauksen URL:n mukana identiteetin tarjoaja lähettää riippuvalle osapuolelle käyttäjän käyttäjätiedot, kuten OpenID:n, jonka riippuva osapuoli tallentaa käyttäjää varten luotuun istuntoon. Muut istunnon aikana tehdyt kyselyt autentikoidaan istunnon avulla. [Chapman and Chapman, 2012]

OpenID on sekä kehittäjän että käyttäjän näkökulmasta helpottava ratkaisu. Käyttäjän ei tarvitse muistaa kuin yksi salasana ja jos salasana unohtuu, riittää, että sen vaihtaa vain yhdellä, turvalliseksi todettua standardia hyödyntävän palveluntarjoajan sivustolla. OpenID helpottaa myös kehittäjää, sillä kehittäjän ei tarvitse ottaa kantaa identiteettiin liitty-

vistä toiminnallisuuksista kuten salasanan tallentamisesta, uusimisesta ja tunnuksen poistamisesta, vaan kaiken tämän voi ulkoistaa toisaalle. [Chapman and Chapman, 2012]

OpenID:ssä on kuitenkin riskinsä, sillä salaamattomalla http-yhteydellä käytettäessä se on erityisen altis mies välissä -hyökkäykselle. Riski kohdistuu vaiheeseen, jossa käyttäjä ohjataan onnistuneen kirjautumisen päätteeksi riippuvan osapuolen sivustolle. Tällöin hyökkääjän tarvitsee saada käsiinsä vain URL, johon käyttäjää ollaan ohjaamassa, minkä avulla hyökkääjä voi tunnistautua riippuvan osapuolen sivustolle rajattomia kertoja. Riskiä on pyritty minimoimaan nonce-avaimen avulla. Tällöin riippuva osapuoli lähettää identiteetin tarjoajalle kerran voimassa olevan noncen, jonka identiteetin tarjoaja palauttaa takaisin riippuvalle osapuolelle autentikaation jälkeen. Nonce on voimassa vain kerran, joten hyökkääjän pitää saada estettyä ohjauksen suorittaminen ainakin siksi aikaa, että hyökkääjä ehtii itse autentikoitumaan riippuvan osapuolen sivustolle. [van Delft and Oostdijk, 2010]

OpenID:n tarjoaman käyttäjätunnuksen vaihtaminen vapauttaa edellisen käyttäjätunnuksen. Jos uusi käyttäjä ottaa jo käytössä olleen käyttäjätunnuksen käyttöönsä ja kirjautuu sivustolle, jonka autentikaatio on toteutettu OpenID:n avulla ja jota käyttäjätunnuksen edellinen haltija on joskus kirjautunut, paljastaa sivusto vanhan käyttäjän tiedot uudelle käyttäjälle [van Delft and Oostdijk, 2010]. Hyökkäyksen tekijä voi myös olla OpenID:n riippuva osapuoli, joka ohjaa käyttäjän palveluntarjoajan sivustoa muistuttavalle sivustolle ja kaappaa käyttäjän syöttämät tunnukset [van Delft and Oostdijk, 2010].

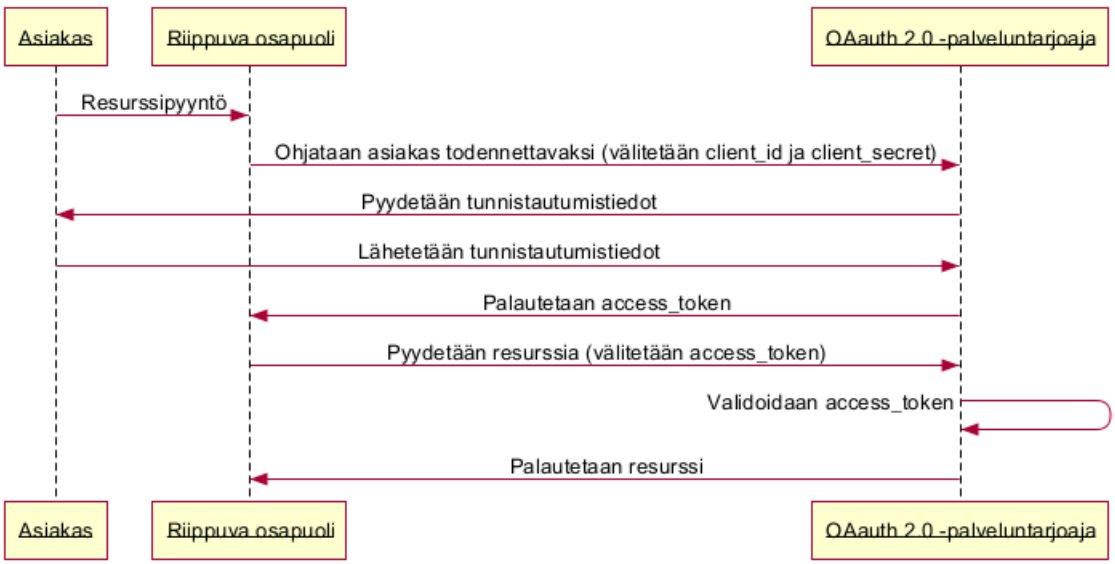
OpenID:tä voidaan pitää myös yksityisyyden kannalta kyseenalaisena ratkaisuna, sillä jokainen istunto aloitetaan aina tunnistautumalla kolmannen osapuolen sivustolla. Identiteetin tarjoaja pystyy siis helposti tarkkailemaan, mitä palvelua rekisteröitynyt käyttäjä kulloinkin käyttää [van Delft and Oostdijk, 2010]. OpenID:hen liittyvien riskien takia esimerkiksi Google on lopettanut OpenID-tukensa elokuussa 2015 [Li and Mitchell, 2015].

3.5. OAuth 2.0

OAuth 2.0. on token-perustainen autorisointistandardi, jossa käyttäjätunnuksen ja salasanan sijaan todentaminen perustuu tokeneihin, eli tarkoitusta varten generoituihin avaimiin. Siinä missä todentamisen tarkoituksena on varmistaa, että käyttäjä on se, joka hän väittää olevansa, auktorisoinnin tarkoituksena on varmistetaa, että käyttäjä on oikeutettu resurssiin tai funktioon, jota hän kutsuu. OAuth 2.0:n avulla käyttäjä voi jakaa verkkopalvelulle (riippuva osapuoli) resursseja, jotka hän on ladannut toiseen verkkopalveluun (resurssin tarjoaja) ilman, että joutuu luovuttamaan käyttäjätunnusta tai salasanaansa riippuvalle osapuolelle.

Jotta riippuva osapuoli voi hyödyntää OAuth 2.0 -palveluntarjoajan resursseja, tulee kehittäjän rekisteröityä palveluntarjoajan sivustolle ja luovuttaa uudelleenohjaus-URL:n, johon käyttäjä onnistuneen tunnistautumisen jälkeen ohjataan, sekä usein myös autorisointialan (scope), jonka avulla resurssin tarjoajalle kerrotaan resurssit, joita riippuva osa-

puoli haluaa hyödyntää. Rekisteröinnin suoritettuaan riippuva osapuoli saa asiakastunnuksen (client ID) ja asiakassalaisuuden (client secret).



Kuva 4: OAuth 2.0 -sekvenssikaavio

Kuvan 4 OAuth 2.0 toimii seuraavasti: Kun riippuva osapuoli haluaa hyödyntää resurssin tarjoajan resursseja, käyttäjä ohjataan asiakastunnuksen ja -salaisuuden kanssa autentikoitumaan resurssin tarjoajan sivustolle. Koska yksi resurssin tarjoaja voi ylläpitää yhden tunnuksen takana useita erilaisia palveluita, kerrotaan käyttäjälle autorisointialan perustella ennen autentikointia, mihin resursseihin riippuva osapuoli haluaa saada pääsyn. Kun todentaminen on onnistunut, resurssin tarjoaja ohjaa käyttäjän takaisin rekisteröinnin yhteydessä luovutettuun uudelleenohjaus-URL:iin. Ohjauksen yhteydessä riippuvalle osapuolelle välitetään autorisoinnin mahdollistavaa dataa kuten pääsyavain (access token), minkä avulla resurssien hakeminen resurssin tarjoajalta onnistuu sekä päivitysavain (refresh token), jonka avulla vanhentunut pääsyavain voidaan tarpeen tullen uusia. [Chapman and Chapman, 2012]

Vaikka OAuth 2.0 on tarkoitettu resurssien autorisointiin, sitä voidaan suositusten vastaisesti käyttää myös käyttäjän todentamiseen. Koska resurssi itsessään liittyy usein yksittäiseen käyttäjään, on luonnollista, että myös käyttäjätietoja käsitellään resursseina. OAuthia autentikaatioon hyödyntävä riippuva osapuoli pyrkii hyödyntämään siis pelkkää käyttäjäresurssia, jolloin se pyytää resurssin tarjoajalta esimerkiksi pelkän sähköpostiosoitteen käyttäjätunnisteeksi ja liittää tunnistetta vastaavan käyttäjän esimerkiksi asiakkaan ja palvelimen välillä kulkevaan istuntoon.

Richer [2014] osoittaa, ettei OAuth 2.0 ole missään tapauksessa suositeltu tai turvallinen tapa todentaa käyttäjä. OAuth 2.0 -perustaisessa todentamisessa autorisaatiopalvelimen palauttamaa access_tokenia saatetaan virheellisesti käyttää todisteena onnistuneesta

autentikoinnista. Access_token ei kuitenkaan ole riippuvan osapuolen purettavissa, eikä sen verifiointi ole myöskään mahdollista [Richer, 2014]. Riippuva osapuoli ei siis pelkän access_tokenin avulla voi päättää, onko käyttäjä todella kirjautunut sisään.

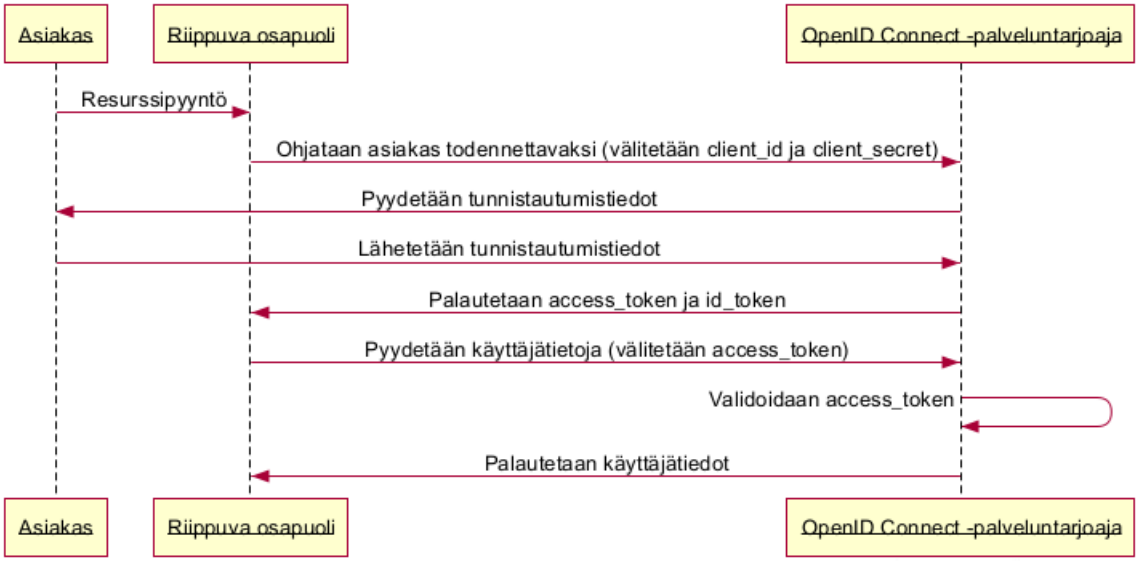
Myös käyttäjäresurssin hyödyntäminen käyttäjän todentamisessa on ongelmallinen keino. Jos kirjautumisen yhteydessä vaihdettu access_token palauttaa oikean käyttäjän tiedot, voi riippuva osapuoli hakea uuden access_tokenin ja siten myös saman käyttäjän tiedot uudelleen refresh_tokenin avulla. Koska refresh_token ei välttämättä vanhene koskaan, ei käyttäjän tarvitse kirjautua järjestelmään kuin yhden kerran. Riippuva osapuoli hakisi siis käyttäjän tiedot automaattisesti refresh tokenin avulla eikä käyttäjän kirjautuminen käytännössä vanhenisi koskaan. [Richer, 2014]

OAuth 2.0 ei myöskään pakota hyödyntämään nonce-avainta, jonka avulla riippuva osapuoli voisi varmistaa, että kutsuttu palveluntarjoaja on se, jolta myös autorisaatiovastaus tulee. Hyökkääjä voisi tässä tapauksessa ujuttaa keinotekoisen access token -vastauksen sekä samalla myös keinotekoisen vastauksen käyttäjäresurssikutsuun ja näin saada riippuvan osapuolen luulemaan, että hyökkääjän esittämä käyttäjä olisi kirjautunut sisään. [Richer, 2014]

OAuth 2.0 -standardi ei myöskään pakota palveluntarjoajaa palauttamaan käyttäjäresurssia missään tietyssä muodossa. Jos riippuva osapuoli tarjoaa käyttäjän todentamiseen useaa eri OAuth 2.0 -palveluntarjoajaa, saattaa käyttäjäresurssin muoto vaihdella ja esimerkiksi palveluntarjoaja a palauttaisi käyttäjätunnisteen avaimella user_id ja palveluntarjoaja b saman tunnisteen avaimella sub. Tällöin jokaista palveluntarjoajaa varten jouduttaisiin varautumaan kustomoidulla ohjelmakoodilla, jotta käyttäjät voisivat hyödyntää mahdollisen montaa eri OAuth 2.0 -palveluntarjoajaa identiteettinsä todistamiseen. [Richer, 2014]

3.6. OpenID Connect

OpenID Connect on vuonna 2014 julkistettu OAuth 2.0:n päälle rakennettu avoin autentikointistandardi, jonka tarkoituksena on ratkaista OpenID:hen liittyviä turvallisuus- ja käytettävyysongelmia sekä laajentaa turvalliseksi ja käytettävyydeltään miellyttäväksi miellettyä OAuth 2.0 -standardia käyttäjän todentamiseen. OpenID Connectin tarpeesta ja toisaalta OpenID 2:n ongelmista kertoo se, että vaikka standardi julkaistiin vuonna 2014, oli valmiita OpenID Connectia hyödyntäviä toteutuksia jo runsaasti ennen varsinaista julkaisuajankohtaa [Li and Mitchell, 2015].



Kuva 5: OpenID Connect -sekvenssikaavio

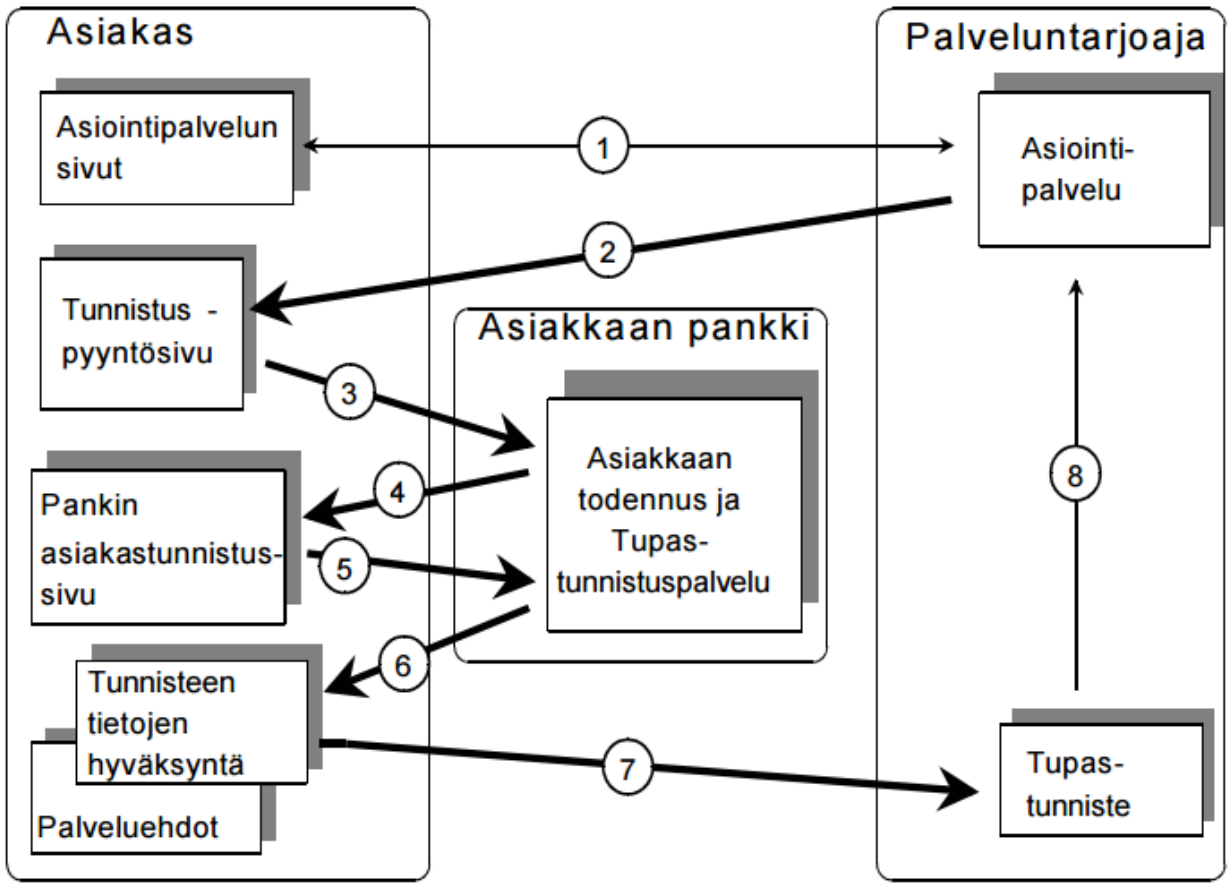
Kuvassa 5 esitetty OpenID Connect -standardi toimii hyvin samankaltaisesti kuin OAuth 2.0. Todentamisen kannalta olennaisin lisä on OpenID Connect -palveluntarjoajan access_tokenin lisäksi palautettava id_token. id_token on JSON Web Token, joka on riippuvan osapuolen validoitavissa ja joka access_tokenista poiketen sisältää jo itsessään käyttäjän todentamiseen vaadittavaa dataa, kuten yksilöidyn käyttäjätunnuksen, OpenID Connect -palveluntarjoajan tunnuksen, riippuvan osapuolen tunnuksen sekä tokenin voimassaoloajan. id_token on myös allekirjoitettu avaimella, joka lähetetään OpenID Connect -palveluntarjoajalle samassa kutsussa kuin missä asiakas ohjataan todennettavaksi.

Palveluntarjoajan palauttama access token on sinänsä autentikaation kannalta turha, sillä käyttäjä voidaan todentaa jo pelkän id_tokenin avulla. Access_tokenin avulla riippuva osapuoli voi kuitenkin halutessaan laajentaa käyttäjädataansa OpenID Connect -palveluntarjoajan tarjoamalla käyttäjädatalla, minkä takia riippuvan osapuolen datamäärää voidaan keventää.

Koska OpenID Connect ei pysty pakottamaan riippuvan osapuolen kehittäjää noudattamaan standardiaan, jää turvallisen autentikoinnin varmistaminen lopulta sitä hyödyntävän kehittäjän vastuulle. Kuten Li ja Mitchell [2015] osoittavat Googlen kehittämän OpenID Connect -palveluntarjoajan tapauksessa, OpenID Connectin käyttö ei takaa turvallista todentamista, vaan se voi altistaa sovelluksen muun muassa CSRF- ja mies välissä -hyökkäyksille sekä istuntojen kaappaamiselle. Riskit voidaan kuitenkin minimoida noudattamalla OpenID Connect -palveluntarjoajan, eli tässä tapauksessa Googlen, omia oppaita ja huolehtimalla erityisesi, että kaikkein komponenttien välillä kulkevat HTTP-yhteydet ovat SLL/TLS-salattuja.

3.7. Tupas-tunnistuspalvelu

Pankkien Tupas-tunnistuspalvelu on suomalaisten pankkien yhdessä määrittelemä palvelu, jonka avulla verkkopalveluiden tuottaja voi todentaa käyttäjänsä Tupas-tunnisteiden avulla [Finanssialan keskusliitto, 2013b]. Tupas-tunnistuspalvelua hallinnoi, ylläpitää ja kehittää Finanssialan keskusliitto. Tupas-tunnistaminen täyttää Suomen laissa määritellyt kriteerit vahvasta sähköisestä tunnistamisesta [Finlex, 2009].



Kuva 6: Tupas-tunnistuspalvelu [Finanssialan keskusliitto, 2013a]

Kuvassa 6 on esitetty Tupas-tunnistuspalvelu toiminta sekvenssikaaviota muistuttavan kaavion avulla. Tupas-palvelua hyödyntävä ohjelmisto antaa käyttäjän ensin valita oman pankkinsa ja uudelleenohjaa käyttäjän valitun pankin tunnistamissivulle, jossa käyttäjän tulee kirjautua omilla tunnuksillaan sisään. Uudelleenohjauksen mukana pankille välitetään tunnistamispyyntö, jonka sisältö ja palveluehdot esitetään käyttäjälle onnistuneen tunnistamisen jälkeen. Käyttäjä voi joko hyväksyä tai hylätä pyynnön. Kun tunnistamispyyntö on hyväksytty, pankki muodostaa vastaussanoman eli Tupas-tunnisteen, ja käyttäjä ohjataan takaisin palveluntarjoajan sivustolle. Tupas-tunniste välitetään palvelun-

tarjoajalle ohjauksen mukana ja palveluntarjoajan on verifioitava tunniste. [Finanssialan keskusliitto, 2013a]

Palveluntarjoajan pankille lähettämä tunnistepyyntö on tarkkaan määritetty kokonaisuus, joka lähetetään pankille HTML-lomakkeen piilotettuina muuttujina POST-pyynnöllä. Tunnistepyyntö sisältää muun muassa palveluntarjoajan tunnuksen, palvelun kielen, pyynnön ajankohtaan perustuvan yksilöidyn tunnisteen, paluuosoitteen, peruutusosoitteen, hylkäysosoitteen, tarkisteen sekä salaukseen käytetyn algoritmin tunnuksen. [Finanssialan keskusliitto, 2013a]

Pankin generoima Tupas-tunniste palautetaan palveluntarjoajalle URL:n kyselyparametreihin sisällytettynä. Tupas-tunniste sisältää muiden muassa tunnisteen yksilöidyn tunnisteen, joka sisältää myös tiedon tunnisteen luodusta pankista, tunnistetun asiakkaan nimen, salaukseen käytetyn algoritmin tunnuksen sekä asiakkaan yksilöintitiedon, joka voi olla joko selväkielinen asiakastunnus tai jokin muu salattu yksilöintitieto.

Pankin palauttaman yksilöintitiedon perusteella palveluntarjoaja voi luoda asiakkaasta esimerkiksi uuden käyttäjän järjestelmään tai yhdistää tunnisteen jo olemassa olevan käyttäjän tietoihin. Käyttäjää varten voidaan tämän jälkeen luoda esimerkiksi istunto, jonka avulla muut käyttäjän palvelimelle lähettämät pyynnöt voidaan todentaa.

Finanssialan keskusliiton [2013a] mukaan kuvan 6 numeroiduista tapahtumista 2-7 tulee salata SSL-salausta hyödyntäen, jotta transaktion sisältö ei ole ulkopuolisen luettavissa esimerkiksi mies välissä -hyökkäyksen avulla. Palveluntarjoajan palvelinohjelmiston tulee tukea 128 bitin avaimilla toteutettua SSL-salausta, vaikka salausavaimen pituus määräytyykin käyttäjän käyttämän selaimen ominaisuuksien perusteella.

Sekä tunnistuspyyntö että itse tunniste suojataan ennalta määritellyllä tarkisteavaimella, jonka avulla transaktion data on verifioitavissa ja jolloin tunnisteen tai tunnistepyynnön tietoja ei voi muuttaa ilman, että tiedon vastaanottaja huomaisi muutokset. Tarkisteavain eli MAC-avain on satunnaisesti generoitu vähintään 32 merkkiä pitkä merkkijono, jonka generoimiseksi pankki toimittaa vaihtuvia attribuutteja. [Finanssialan keskusliitto, 2013b]

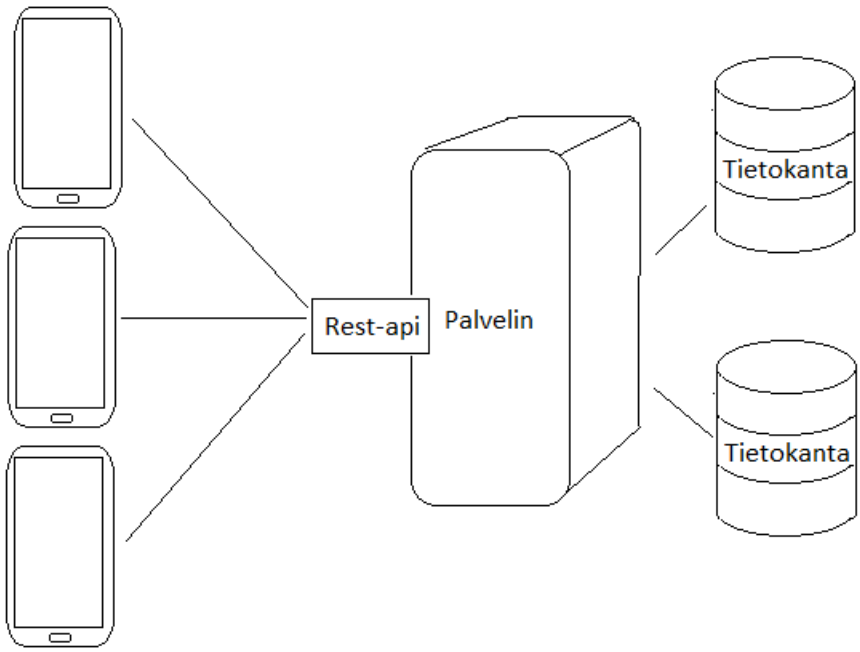
Tupas-tunnistaminen vaatii Finanssialan keskusliiton [2013a] mukaan pankkitunnusten ja avainten hallinnointiin liittyvien toimintojen jäljitettävyyden, joten kaikki epäilyttävä toiminta tulisi olla pankkien sekä Tupas-tunnistamista hyödyntävien palveluntarjoajien lokeissa kirjattuna. Tallennettuja lokitietoja tulee säilyttää viranomaismääritysten mukaisen vähimmäisajan, jonka jälkeen kaikki lokeihin tallennetut henkilötiedot on lain mukaan hävitettävä.

Tupas-tunnistaminen ei salli kertakirjautumista eli kirjautumista usean palveluntarjoajan sivustolle yhdellä tunnistamiskerralla. Sen sijaan istunnon siirrosta voidaan sopia Finanssialan keskusliiton määrittelemien edellytysten mukaisesti. Istunnon siirtämisen vaatimuksena on, että käyttäjä kirjautuu ensimmäisen palveluntarjoajan palvelusta ensin ulos ennen seuraavaan siirtymistä.

Tupas ei ota kantaa siihen, kuinka palveluntarjoaja todentaa käyttäjän pyynnöt Tupas-tunnistamisen jälkeen, joten Tupas-tunnistamisen käyttöönotto ei yksistään riitä järjestelmän autentikaatiomekanismiksi. Järjestelmään tulee siis erikseen rakentaa mekanismi, jonka avulla voidaan varmistaa, että Tupas-tunnistuksen jälkeen tehdyt kutsut tulevat yhä samalta henkilöltä, joka on suorittanut itse pankkitunnistautumisen.

4. REST

Kuvassa 7 esitelty REST, eli Representational State Transfer, on web-palveluiden kehittämiseen suunniteltu arkkitehtuurimalli, jonka nimesi ja esitteli Roy Fielding väitöstutkimuksessaan *Architectural Styles and the Design of Network-based Software Architectures* vuonna 2000. REST on asiakas-palvelin-mallin päälle rakennettu arkkitehtuuri, joka toimii palvelimen tarjoamana rajapintana asiakkaan ja palvelimen välissä pyrkien palvelemaan yhdenmukaisesti sekä asiakasryhmän koon että asiakassovellusten ja -laitteiden kannalta mahdollisimman laajaa asiakasjoukkoa [Massé, 2012].



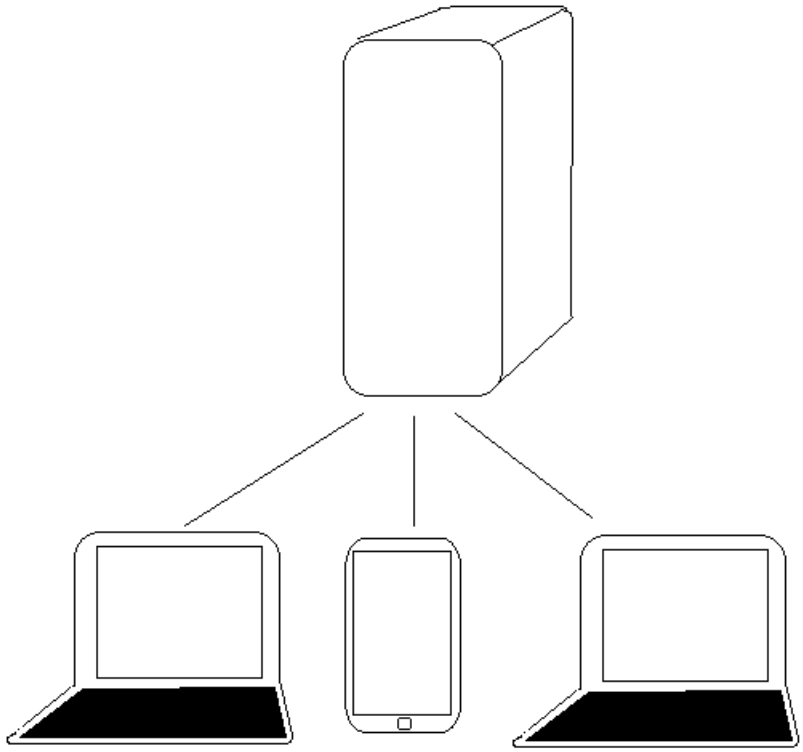
Kuva 7: REST-rajapinta

Koska REST on HTTP-standardin päälle rakennettu, sen toimintaperiaatteet ja rajoitukset liittyvät läheisesti kuuteen Fieldingin jo vuonna 1993 esittelemään web-arkkitehtuurin rajoitukseen [Fielding, 2000; Masse, 2012]:

- Asiakas-palvelin-malli (Client-Server)
- Tilattomuus (Stateless)
- Välimuisti (Cache)
- Yhdenmukainen rajapinta (Uniform Interface)
- Kerroksista koostuva järjestelmä (Layered System)
- Suoritettavien ohjelmien palauttaminen palvelimelta asiakkaille (Code-On-Demand).

Asiakas-palvelin-mallissa (kuva 8) palvelinkomponentti palauttaa sille tehtyjen kyselyjen perusteella vastauksia asiakaskomponentille [Fielding, 2000]. Palvelimen palautta-

mat vastaukset voivat sisältää sisällön (body) lisäksi myös metatietoa vastauksesta, kuten informaatiota palvelimelle tehdyn pyynnön hylkäämisestä ja hyväksymisestä, sisällön tyypistä ja pyynnön tehneestä asiakkaasta. Asiakas—palvelin-malli ei ota kantaa järjestelmän tekniseen toteutukseen niin kauan, kun järjestelmä täyttää vaatimuksen internetin yhdenmukaisuudesta [Masse, 2012].



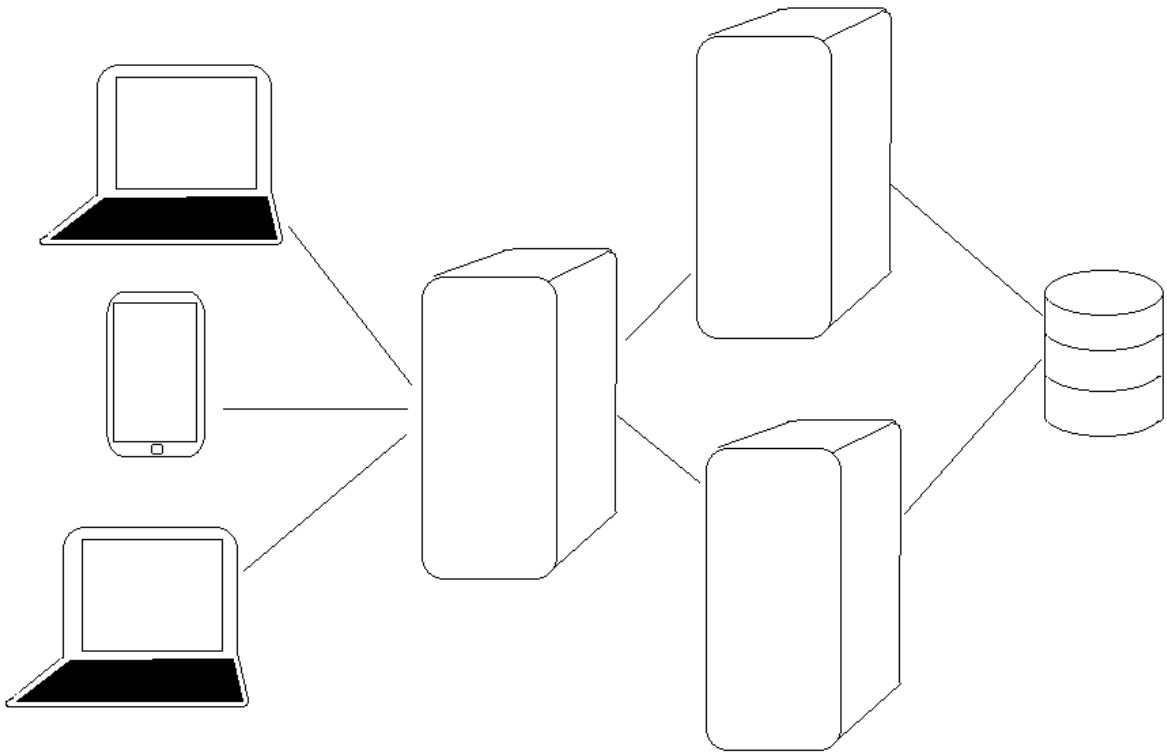
Kuva 8: Asiakas-palvelin-malli

Tilattomuudella tarkoitetaan nimenomaan palvelimen tilattomuutta. Kakki palvelimen tarvitsema tieto kulkee kyselyn mukana. Palvelimen tilattomuus on ratkaisu palvelimen kuormituksen siirtämisellä yksittäisen asiakkaan vastuulle, jolloin palvelin voi palvella mahdollisimman montaa asiakasta [Masse, 2012]. Ratkaisu ei kuitenkaan ole täydellinen, sillä saman tiedon siirtäminen edestakaisin kuormittaa palvelimen ja asiakkaan välistä kaistaa [Fielding, 2000].

Välimuisti on keino tallentaa ja hakea usein väliaikaseksi tarkoitettua ja pienen ajan sisällä hyödynnettävää tietoa mahdollisimman nopeasti. Välimuisti voi fyysisesti sijaita asiakkaan tai palvelimen puolella, mutta REST-rajapinnan tapauksessa välimuistilla tarkoitetaan nimenomaan asiakkaan puolelle ylläpidettävää välimuistia. Välimuistilla voidaan vähentää toistuvien kyselyiden määrää ja siten minimoida verkkoyhteyden kuormitus. REST-rajapinta palauttaa asiakassovellukselle resurssin mukana tiedon siitä, onko resurssi tallennettavissa välimuistiin (cacheable). [Fielding, 2000]

REST-rajapinta tarjoaa jokaiselle resurssille yhdenmukaisen rajapinnan, mikä mahdollistaa järjestelmän yhdenmukaisuuden, yleispätevyyden ja näkyvyyden. Standardoitu rajapinta rajoittaa järjestelmän tehokkuutta, sillä yksinkertaisinkin kysely ja vastaus tulee prosessoida ja muodostaa määritellyllä tavalla. [Fielding, 2000]

Järjestelmän rakentaminen kerroksista parantaa järjestelmien luotettavuutta, turvallisuutta, evoluutiota ja skaalautuvuutta. Kuten kuvassa 9 nähdään, yksittäinen kerros tietää vain itseään lähimmistä kerroksista. Tällöin esimerkiksi asiakkaana toimiva yksittäinen tietokone ei pääse suoraan manipuloimaan tietokannassa sijaitsevaa resurssia, vaan asiakkaan ja tietokannan välissä palveleva rajapinta tarjoaa siihen rajoitetut ja tarvittaessa yksilöidyt keinot. Järjestelmän rakentaminen kerroksista myös helpottaa järjestelmän ylläpitoa, sillä hyvin suunniteltuna yksittäisen komponentin muuttamisen tiedetään aiheuttavan muutoksia vain sitä lähimpiin komponentteihin. Kerroksellisuus auttaa myös suojamaan vanhoja ja mahdollisesti haavoittuneita komponentteja. [Fielding, 2000]



Kuva 9: Kerroksista koostuva järjestelmä

REST-rajapinta voi tarjota interaktiivisuuden lisäämiseksi asiakkaalle myös lisäosia, sovelmia (applets) tai komentosarjoja (scripts). Toisin kuin muut Fieldingin esittämät web-arkkitetuuria rajaavat keinot, on Code-On-Demand-rajoitus vapaaehtoinen laajennos. [Masse, 2012]

REST-rajapinnan ydinkomponentti on resurssi. Resurssi on informaation abstraktio, jolla on sisältö ja representaatio. Sisältö voi olla esimerkiksi tietoa reaalimaailman objektista, kuten henkilöstä tai vaikkapa ajoneuvosta. Representaatiolla tarkoitetaan muotoa tai tapaa, jolla sisältö on kuvattu. Resurssin representaatio voi olla esimerkiksi HTML-dokumentti, JPG-kuva tai JSON-muotoinen olio. Kukin resurssi on identifioitu yksiselitteisesti esimerkiksi URI:n (Uniform Resource Identifier) avulla. Sekä resurssiin että sen representaatioon liittyy metadataa, joka sisältää tietoa resurssin representaatioformaatista, aikaleimoja esimerkiksi luonti- ja muokkauspäivämäärästä sekä linkin resurssiin itseensä. [Fielding, 2000]

REST-arkkitehtuuriin liittyy läheisesti tila ja tilattomuus. REST-rajapintaa hyödyntävällä asiakkaalla samoin kuin jokaisella resurssilla on tila. Resurssiin kohdistettu kutsu voi muokata resurssin tai asiakkaan tilaa. Esimerkiksi HTTP-metodeista GET muokkaa vain asiakkaan tilaa, kun taas POST, PUT ja DELETE saattavat muuttaa sekä resurssin että asiakkaan tilaa. REST:n tapauksessa tilattomuudella viitataan palvelimen tilaan. Palvelimen toiminta on suunniteltu niin, että se käsittelee jokaisen pyynnön yksitellen erillään toisistaan, eikä mikään pyyntö saa olla riippuvainen toisesta pyynnöstä. Tilattomuuden takia minkäänlaisia palvelimen puolelle tallennettuja istuntoja ei siis tule käyttää, vaan pyyntö itsessään sisältää kaiken tarvittavan tiedon onnistuakseen. Palvelimen ei tarvitse ylläpitää erillistä välimuistia tai tietokantatoteutusta, mikä nopeuttaa kyselyiden käsittelyä.

Tilattomuuden ansiosta myös järjestelmän hajauttaminen on helpompaa ja samasta palvelusta voi yhtä aikaa olla olemassa monta eri instanssia, joista kaikki pystyvät vastaamaan kutsuihin samalla tavalla. Monen instanssin olemassa oleminen mahdollistaa kuormituksen tasaamisen load balancer -palvelun avulla, jolloin erillinen komponentti tarkistaa heti kyselyn alussa, mikä palvelun instansseista on kulloinkin vähiten kuormittunut ja ohjaa käyttäjän kyselyt vastauksen perusteella oikeaan paikkaan.

4.1. REST-rajapinnan käyttöliittymä

REST-rajapinta hyödyntää tarjoamiensa resurssien hakemiseen, lisäämisen, poistamiseen ja muokkaamiseen HTTP-standardissa määriteltyjä GET-, POST-, DELETE- ja PUT-metodeita. GET on metodeista yleisin ja sitä käytetään tiedon, kuten yksittäisen resurssin sisällön hakemiseen tai esimerkiksi tietyntyyppisten resurssien listaamiseen. GET-kyselyn laajuutta voidaan rajata kyselyparametrien avulla, jolloin URL:n perään liitetään avain-arvo-pareja. Esimerkiksi URL:n http://www.kaakeli.net/laatat?pituus=15&leveys=20 avulla kuvitteellisesta kaakeli.net-palvelusta voidaan hakea laattoja, joiden pituus on 15 cm ja leveys 20 cm. Yksittäisen laattaresurssin GET-URL voisi olla http://www.kaakeli.net/laatat/5252c435df, jossa arvolla ”5252c435df” tarkoitetaan kyseisen resurssin yksilöityä tunnistetta. [Masse, 2012]

POST-metodia käyttäen palvelimelle voidaan luoda uusi resurssi. Uuden resurssin tila eli sisältö voidaan lähettää POST-pyynnön body-osassa esimerkiksi avain-arvo-pareina

(form-data- tai x-www-form-urlencoded-muodossa) tai JSON-objektina. [Masse, 2012]. DELETE-metodi on nimensä mukaisesti tarkoitettu resurssin poistamiseen. DELETE-metodia käytetään samaan tapaan kuin GET-metodia yksittäisen resurssin yhteydessä. Kuvitteellisessa kaakeli.net-palvelussa kaakelin poistaminen voisi tapahtua DELETE-kutsulla, jonka URL olisi http://www.kaakeli.net/laatat/5252c435df.

PUT-metodilla muokataan jo olemassa olevan resurssin tilaa. PUT-kutsun yhteydessä palvelimelle lähetetään avain-arvo-pareina tieto resurssin uudesta tilasta. PUT-metodin mukana lähetettävä data voi sisältää vain ne arvot, joita kutsulla halutaan muuttaa tai kutsun mukana voidaan lähettää resurssin tila kokonaisuudessaan eli esimerkiksi siinä muodossa, jossa palvelin on palauttanut yksittäisen resurssin GET-kutsun yhteydessä. [Masse, 2012]

HTTP-metodit voidaan jaotella sen mukaan, ovatko ne idempotentteja tai turvallisia. Idempotentilla metodilla tarkoitetaan metodia, joka tuottaa samanlaisen vastauksen riippumatta metodin toistokerroista. HTTP-metodeista idempotentteja ovat GET, PUT ja DELETE. Turvallisella metodilla tarkoitetaan metodia, jonka käyttäminen ei aiheuta muutosta metodin kohteena olevaan resurssiin. REST-rajapinnan hyödyntämistä metodeista vain GET on termin merkityksen mukaisesti turvallinen. [Amundsen and Richardson, 2013]

Käytännössä metodin turvallisuus ja idempotenttius tulee ottaa huomioon niin, että GET ei milloinkaan saa tehdä muutoksia resursseihin ja toisaalta niin, että ei-idempotentilla metodilla ei tule tehdä tehtyä toista kutsua samaan resurssiin ennen kuin tieto epäonnistuneesta kutsusta saapuu perille. Koska esimerkiksi DELETE, eli resurssin poistava metodi, on idempotentti, ei DELETE-muotoisen pyynnön lähettäminen uudelleen ole riskialtista. Sen sijaan toistamalla POST-tyyppistä pyyntöä luodaan uusia resursseja täsmälleen toistojen lukumäärän verran, mikä saattaa aiheuttaa sovelluksen sisäisiä ongelmia. [Masse, 2012]

Myös palvelimen tuottamat vastaukset muodostetaan HTTP:n standardissa kuvattuja tapoja noudattaen. Vastaus (response) tulee sisältää aina metatietoa, kuten vastauksen sisällön formaatin, vastauksen statuskoodin sekä URL:n, jota kutsumalla vastaus on muodostettu. Vastauksen formaatti on tärkeä osa metatietoa, jotta asiakassovellus osaa hyödyntää saamansa datan oikein. Vääränmuotoinen data aiheuttaa asiakassovellukselle todennäköisesti virheen tai poikkeuksen. [Masse, 2012]

Statuskoodi	Selite	Kuvaus
200	Ok	Pyynnön suorittaminen onnistui
201	Created	Uuden resurssin luominen onnistui
202	Accepted	Pyynnön aikaansaama asynkroninen suoritus on alkanut
301	Moved Permanently	Resurssin URL on muuttunut
400	Bad request	Määrittelemätön kutsun epäonnistuminen
401	Unauthorized	Resurssi vaatii autentikoinnin, mutta käyttäjätunnuksia ei ole annettu tai niiden lukeminen ei onnistunut
403	Forbidden	Käyttäjän autentikointi on onnistunut, mutta kyseisellä käyttäjällä ei ole pääsyä pyydettyyn resurssiin
404	Not found	URL on oikeanlainen, mutta haluttua resurssia ei löytynyt
500	Internal Server Error	Normaalisti automaattinen vastaus, jonka aiheuttaa virhe tai poikkeus pyynnön suorittamisessa

Taulukko 1: HTTP-standardissa kuvatut statuskoodit

Yleisimmät statuskoodit ja niiden kuvaukset on esitelty taulukossa 1. Statuskoodi sisältää jo itsessään tiedon vastauksen onnistumisesta tai epäonnistumisesta, joten asiakassovellukselle saattaa usein riittää tieto pelkästä statuskoodista, eikä muuta REST-rajapinnan palauttamaa informaatiota edes tarvita. Epäonnistuneet statuskoodit alkavat numeroilla neljä tai viisi ja onnistuneet tai lisätoimenpiteitä vaativat numeroilla 1, 2 tai 3. Eri statuskoodit on tarkasti määritelty sen mukaan, minkälaisen tuloksen kutsu palvelimella lopulta aiheutti. [Masse, 2012]

REST-rajapinnalle kohdistetut kyselyt muodostetaan palvelimen osoitteesta sekä kauttaviivoin erotetuista substantiiveista kuten http://www.kaakeli.net/myyjat tai http://www.kaakeli.net/myyjat/5252c435df/asiakkaat. Yksittäiseen resurssiin viitataan yksilöivän tunnisteen avulla ja listatessa käytetään resurssijoukon nimen monikkoa. CRUD-kyselyiden (Create, Read, Update, Delete) URL:ssa tulee välttää verbejä, kuten http://www.kaakeli.net/hae-laatat tai http://www.kaakeli.net/laatat/5252c435df/poista, sillä metodien nimet ovat jo itsessään riittävän kuvaavia. Jos tietyn resurssin tilaa muutetaan niin, että muuttamisessa käytetään esimerkiksi selkeästi nimettyä funktiota, voidaan funktion nimi lisätä suoraan URL:iin, esimerkiksi http://www.kaakeli.net/myyjat/5252c435df/aseta-lomalle. [Masse, 2012]

Kun resurssit on nimetty riittävän kuvaavasti ja rajapinta on määritelty yhdenmukaisesti ja yleisiä käytäntöjä noudattaen, paranee rajapinnan käytettävyys merkittävästi. Par-

haimmillaan rajapinnan dokumentaatioksi saattaa riittää pelkästään rajapinnan tarjoamien päätepisteiden (end points) listaaminen.

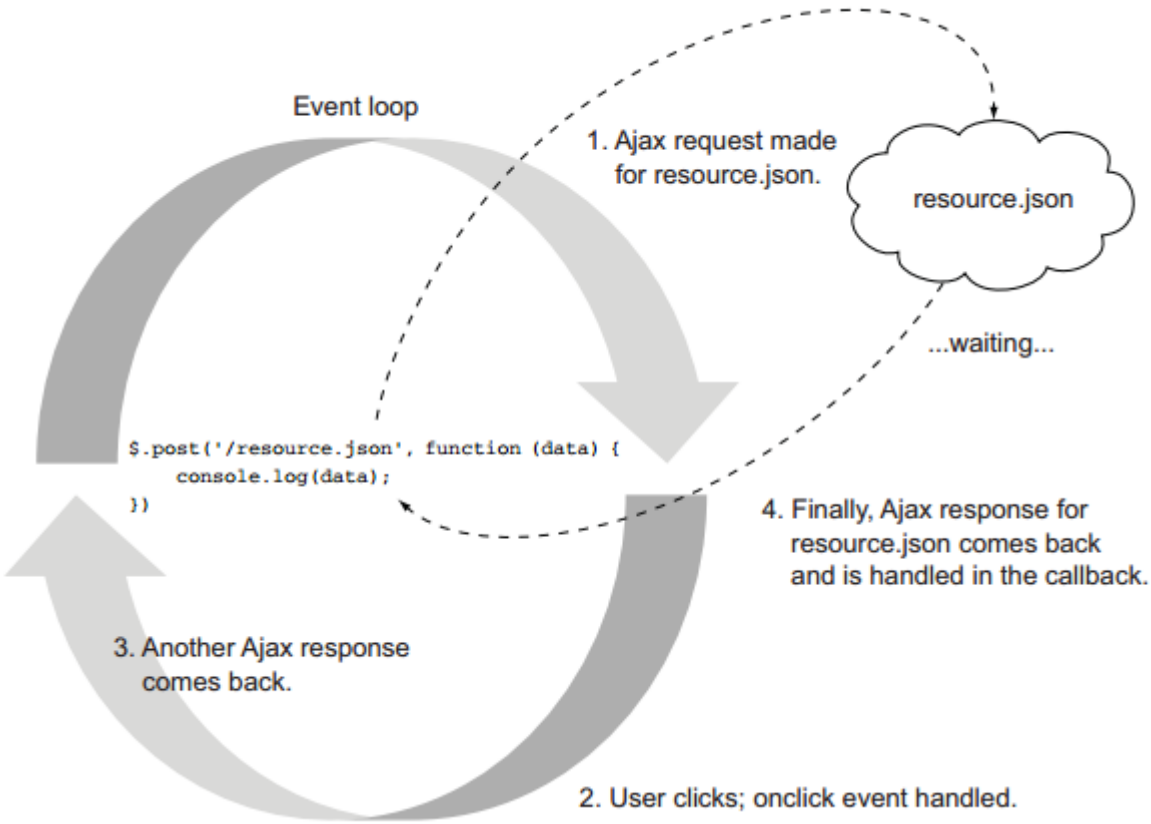
REST-rajapintojen tarjoaman datan yleisimmäksi formaatiksi on muodostunut JavaScript Object Notation. JSON on juuri riittävän kuvaava kuvaamaan resursseja, jotka pääasiassa sisältävät avain—arvo-pareja, mutta tarvittaessa sitä voidaan syventää rajattomasti. JSON:n avulla voidaan siis helposti kuvata resursseja, jotka sisältävät resursseja, jotka sisältävät resursseja ja niin edelleen. JSON on syntaksiltaan hyvin yksinkertainen ja siten myös helppokäyttöinen. JSON-tuki löytyy myös poikkeuksetta jokaisesta suositusta verkko-ohjelmointikielestä. REST-rajapinta voi JSON:in rinnalla myös tukea XML:ää tai HTML:ää, mutta ensisijaisena formaattina JSON on kuitenkin suositeltavin. [Masse, 2012]

5. Demonstraatiosovellus

REST-rajapinnan toteutusta demonstroidaan tutkielmassa yksinkertaisella kurssi-ilmoittautumisapplikaatiolla. Sovelluksen REST-rajapinnan tarjoava palvelinosuus on toteutettu Sails.js-ohjelmistokehyksen sekä MongoDB-tietokannan avulla ja rajapintaa hyödyntävä asiakaspuoli Angular 2 -ohjelmistokehyksen ja Angular 2 -kehitystä nopeuttavan angular2-seed-lisäosan sekä sovelluksen ulkonäköä kohentavan Foundation-kirjaston avulla.

5.1. Node.js, Express.js ja npm lyhyesti

Node.js on ympäristö, joka on suunniteltu palvelinpuolen sovelluksien toteuttamiseen JavaScriptillä. Node.js perustuu Google Chromen V8-moottoriin. JavaScriptin tapaan myös Node.js on tapahtumaohjattu (event driven) ja asynkroninen. Tapahtumaohjautuvuudella tarkoitetaan toiminnollisuutta, jossa tapahtumia lähetetään ja vastaanotetaan tapahtumasyklistä (event loop) käsin. Toisin kuin esimerkiksi PHP:ssä, jossa ohjelman suorittaminen eli tapahtumasyklin eteneminen lopetetaan siirrännän (input/output tai I/O) ajaksi, asynkronisessa tapahtumasyklissä ohjelman suoritus jatkuu muilta osin normaalisti. [Cantelon *et al.*, 2014]



Kuva 10: Tapahtuma-sykli ja asynkronisuus [Cantelon et al. 2014]

Kuvassa 10 havainnollistetaan selaimessa ajettavan JavaScriptin toimintaa siirrännän aikana. Selain tekee POST-tyyppisen kutsun, jolla se hakee resource.json-tiedoston sisältöä. Vaikka kutsu tehdään tapahtumasyklistä käsin, operaatio itsessään tapahtuu syklin ulkopuolella. Tapahtumasykli ei tiedä, kuinka kauan operaation suoritus kestää, joten ilman asynkronisuutta selaimen toiminta lakkaisia kunnes vastaus kutsuun saapuu. Asynkronisuuden ansiosta sovellus vastaa operaation aikana edelleen käyttäjän kutsuihin normaalisti. Operaation tuottama vastaus lähetetään takaisin tapahtumasyklin käsiteltäväksi, eli kuvan 10 tapauksessa selaimen konsoliin tulostettavaksi, takaisinkutsun (callback) avulla. Operaatiolle annetaan siis parametriksi funktio, jota operaation suorituksen jälkeen kutsutaan. Esimerkkitapauksessa resource.json-tiedoston sisältö palautetaan takaisin selaimen käsiteltäväksi takaisinkutsufunktion parametrina ja tulostetaan selaimen konsoliin console.log()-funktion avulla. [Cantelon *et al.,* 2014]

Node.js:n paketinhallintatyökalu on npm (node package manager), joka asentuu automaattisesti Node.js-asennuksen yhteydessä. npm sisältää huomattavan määrän erilaisia JavaScript-lisäosia nopeuttamaan sekä backend- että frontend-kehitystä. [Cantelon *et al.,* 2014]

Express.js on minimalistinen Node.js-ohjelmistokehys, joka tarjoaa työkaluja websovellusten yksinkertaisempaan kehittämiseen. Minimalistisuudella tarkoitetaan Express.js:n tapauksessa vaihtoehtoa monoliittisille web-ohjelmistokehyksille, jotka pyrkivät sisältämään mahdollisimman paljon integroituja ominaisuuksia ja joita käytettäessä suositellaan noudattamaan ohjelmistokehyksen kehittäjien suosittelemia käytäntöjä. Express.js on pieni ja se on suunniteltu modulaariseksi ja mahdollisimman helposti laajennettavaksi sekä useita eri ohjelmointityylejä tukevaksi kehykseksi. [Cantelon *et al.,* 2014]

5.2. Sails.js

Sails.js on MVC-arkkitehtuuria (Model-View-Controller) hyödyntävä ohjelmistokehys, joka käyttää Express.js-ohjelmistokehystä muun muassa HTTP-pyyntöjen suorittamiseen. Sails.js on nopeasti käyttöönotettava ja sisältää useita web-kehitystä nopeuttavia ominaisuuksia ja lisäosia, kuten useaa eri tietokantaa tukevan olioperustaisiin tietokantakyselyihin suunnitellun Waterline ORM:n (object related model), sisäänrakennetun pääsynhallinnan sekä kielistämisen (internationalization tai lyhemmin i18n).

Sails.js asennetaan npm:n avulla”npm -g install sails”-komennolla. Komennon -g-lippu tarkoittaa globaaliksi asennettavaa npm-kirjastoa, joka mahdollistaa Sails.js:n tapauksessa sails-komennon käyttämisen komentoriviltä. Uusi Sails.js-projekti luodaan komennolla ”sails new testProject”, jossa testProject on luotavan projektin nimi. Luotu projekti on heti ajettavissa komennolla ”sails lift”. [Sails.js, 2016b]

Kuva 11: Sails.js:n kansiorakenne [Sails.js, 2016a]

Kuvassa 11 on esitetty Sails.js:n kansiorakenne. Projektin palvelinpuolen logiikka, eli käytännössä MVC-mallin kontrollerit ja mallit (controllers- ja models-kansiot) ovat api-kansiossa, jossa säilytetään myös kukin pääsynhallintasääntö (policies-kansio), muotoillut vastaukset (responses-kansio) sekä palvelut (services-kansio). Kansioon assets tallennetaan kaikki asiakassovellukselle sellaisenaan tarjottava tieto kuten kuvat, asiakassovelluksessa ajettavat JavaScript-tiedostot sekä tyylitiedostot. Kansiossa config ylläpidetään kaikki sovelluksen konfiguraatiodata kuten tietokantayhteyteen, kielistämiseen sekä sovelluksen tarjoamiin HTTP-päätepisteisiin liittyvät asetukset. MVC-mallin näkymät tallennetaan projektin views-kansioon ja projektin rakentamiseen, eli toimintoihin, jotka ”sails lift”-komento käynnistää, liittyvät tiedostot tasks-kansioon. [Sails.js, 2016a]

5.3. Angular 2 ja angular2-seed

Angular 2 on beta-kehitysvaiheessa oleva Googlen kehittämä selainpuolen ohjelmistokehys, joka on täysin uudistettu versio suositusta AngularJS-ohjelmistokehyksestä [Google, 2016]. Angular 2 sisältää AngularJS:n tavoin suuren määrän erilaisia ominaisuuksia, kuten URL:ien reittimäisen, HTTP-pyyntöjen tekemiseen tarkoitetun http-luokan, näkymien ja komponenttien välisten muuttujien yhdistämisen (data binding). Angular 2 antaa kehittä-

jälle pitkälti vapaat kädet sovelluksen arkkitehtuurin muodostamiseen, eikä dokumentaatio ota juurikaan kantaa esimerkiksi sovelluksen kansiorakenteeseen. Dokumentaatiossa on dokumentoitu yksittäisiä esimerkkejä kunkin ominaisuuden hyödyntämiseksi sekä abstraktimpia yksityiskohtia kehyksen toiminnollisuudesta. Beta-kehitysvaihe näkyy erityisesti Angular 2:n rajapintadokumentaatiossa (API 2.0 Preview), jonka usea osio sisältää vain tiedon ”Not Yet Documented”. [Google, 2016]

Yksi merkittävimmistä eroista AngularJS:n ja Angular 2:n välillä on TypeScript, joka on ensisijainen Angular 2:n toteutuskieli. TypeScript ei ole suoraan selaimen suoritettavissa, vaan se tulee erikseen kääntää JavaScriptiksi tsc-lisäosan (TypeScript Compiler) avulla. TypeScript mahdollistaa muuttujien vahvan tyypityksen sekä muun muassa Javasta ja C++:sta tutun luokkaperustaisen ohjelmoinnin periytymisineen ja rajapintoineen. [Google, 2016]

Angular2-seed on Angular 2 -kehittämistä ja erityisesti kehittämisen aloittamista nopeuttava lisäosa, joka sisältää Angular 2 -kehityksen kannalta tärkeitä lisäosia, eri ympäristöille (tuotanto, testaus, kehitys ja e2e-testaus) spesifioidut rakentamisprosessit, valmiin kansiorakenteen sekä pienen määrän esimerkkikoodia. angular2-seed:n dokumentaatio sisältää myös tietoa Angular 2 -kehittämisen hyväksi todetuista ja perustelluista käytännöistä. Projektin rakentaminen tapahtuu tehtävien automatisointiin kehitetyn Gulp-lisäosan avulla. [Gechev, 2016]

5.4. Demonstraatiosovelluksen rakenne

Tutkielmaa varten luodussa esimerkkisovelluksessa Sails.js:n oletuskansiorakennetta on muutettu erottamaan selkeämmin asiakas- ja palvelinpuolen lähdekoodin toisistaan. Käytännössä kansiorakenne sisältää yhden ylätason enemmän ja kaikki asiakaspuolen toiminnollisuus eli Angular 2 -applikaatio on siirretty projektin juurikansion sisälle frontend-kansioon ja palvelinpään logiikka eli Sails.js-sovellus frontend-kansion kanssa kansiohierarkian samalla tasolla olevaan backend-kansioon. Ohjelma ajetaan projektin juurikansiosta komennolla ”npm start”, joka ensimmäisenä kääntää TypeScriptin JavaScriptiksi ja rakentaa asiakaspuolen sovelluksen frontend-kansion sisällön pohjalta ja kopioi rakennetun asiakassovelluksen backend-kansion sisällä olevaan assets-kansioon. Asiakaspuolen rakentamisen ja kopioimisen jälkeen komento rakentaa Sails.js-sovelluksen siten, että sovellus tarjoaa käyttäjälle asiakkaaksi rakennetun frontend-sovelluksen Sails.js:n oletus-URL:sta, eli paikallisesti ajaettuna osoitteesta http://localhost:1337/.

Sovelluksen tarjoamat REST-tyyliset päätepisteet esitellään yksitellen juurikansio/backend/config-kansion alla sijaitsevassa routes.js-tiedostossa. routes.js-tiedosto on Node.js-moduuli, joka sisältää moduulin ulkopuolelle näkyvän osuuden JavaScript-oliona eli käytönnössä avain-arvo-pareina module.exports.routes{}-lohkon sisällä. Jos moduuliin halutaan lisätä vain yksityisiä eli vain moduulin sisäpuolelle näkyviä funktioita ja muuttujia, ne voidaan kirjoittaa module.exports-lohkon ulkopuolelle. Koodikatkelmassa 2 on esi-

tetty todentamispäätepisteiden määritykset. Yksittäinen rivi vastaa yhtä päätepistemääritystä avaimenaan HTTP-metodi ja URL, jonka avulla päätepistettä voidaan kutsua sekä arvonaan ohjain-moduuli ja kyseisen ohjaimen funktio, joka päätepistettä kutsumalla suoritetaan. [Sails.js, 2016c]

```
/**
 * Authentication routes
 */
'post /auth/login': 'AuthController.login',
'get /auth/logout': 'AuthController.logout',
```

Koodikatkelma 2: Esimerkkisovelluksen todentamispäätepisteet [Github, 2016]

Koodikatkelmista 3 ja 4 nähdään, kuinka esimerkkisovelluksessa hyödynnetty Sails.js:n pääsynhallinta toimii. config/policies.js-tiedoston sisällä (koodikatkelma 3) eritellään kukin ohjainluokka sekä luokan sisällä olevat funktiot. Kullekin funktiolle voidaan määrittää pääsynhallintasäännöksi true, tietyn pääsynhallintakäytännön (policy) nimi merkkijonona tai useampi käytäntö käytäntöjen nimiä sisältävänä taulukkona. True-arvolla määritelty funktio on avoin kaikille kutsuille. Käytännön nimi on suora viittaus tiedostoon, jossa kyseinen käytäntö sijaitsee. Koodikatkelmassa 4 on isLecturer-käytäntö, joka tarkistaa, onko pyyntöä tekevän käyttäjän rooli Admin tai Lecturer. Jos ehto täyttyy, palvelin suorittaa ohjainfunktion tai etenee seuraavaan käytäntöön. Jos ehto ei täyty, palautetaan forbidden-moduulin avulla statuskoodi 403. [Github, 2016]

```
CourseController: {
  getCourses: true, // Unauthenticated user can get list of courses
  getCourse: 'isAuthenticated', // User must be authenticated to view course details
  enroll: 'isAuthenticated',
  disenroll: ['isAuthenticated', 'isMe'],
  createCourse: ['isAuthenticated', 'isLecturer'],
  deleteCourse: ['isAuthenticated', 'isLecturer'],
  setLecturer: ['isAuthenticated', 'isLecturer'],
  removeLecturer: ['isAuthenticated', 'isLecturer']
}
```

Koodikatkelma 3: Course-ohjaimen pääsynhallinta [Github, 2016]

```
module.exports = function(req, res, next) {
  if (req.user.role === 'Lecturer' || req.user.role === 'Admin') {
    return next();
  }
  return res.forbidden('You are not permitted to perform this action.');
};
```

Koodikatkelma 4: isLecturer-käytäntö [Github, 2016]

Sails.js-ohjaimet ovat myös Node.js-moduuleita. Ohjain on rakennettu avain-arvo -pareista, joissa avaimet ovat funktion nimiä ja arvot itse funktioita. Ohjainfunktio saa parametrinaan Express.js-kehyksestä tutut req- ja res- parametrit (request ja response). req-parametri sisältää kyselyyn liittyvää dataa, kuten kyselyn otsakkeet, pyydetyn päätepisteen URL:n ja tietoa pyynnön tehneestä asiakkaasta. Res-parametri sisältää muun muassa vastauksen muodostamiseen liittyvää dataa ja toiminnollisuuksia ja sen avulla asiakkaalle voidaan palauttaa esimerkiksi pelkkä statuskoodi (res.ok()), uudelleenohjaus (res.redirect()), html-sivu (res.view()) tai JSON-objekti (res.json()). Esimerkkirajapinta palauttaa asiakkaalle yhdenmukaisesti jokaiseen kyselyyn JSON-olion ja statuskoodin. [Sails.js, 2016c]

Malleissa esitellään sovelluksen tietomalli eli resurssit ja niiden tilat sekä assosiaatiot muihin resursseihin. Malleihin voidaan myös lisätä tietomalliin läheisesti liittyviä funktioita, kuten esimerkkisovelluksen User-mallissa oleva salasanan tiivistäminen [Github, 2016]. Resurssin tilalle määritellään nimi sekä tyyppi ja tarvittaessa tieto yksilöllisyydestä, oletusarvosta ja pakollisuudesta. [Sails.js, 2016c]

```
/**
 * Login endpoint
 * @param req.body.password
 * @param req.body.username
 * @returns JSON object
 */
login: function(req, res) {
  return User.findOne({
      email: req.body.email
    })
    .then(function(user){
      // does user exist and is given password valid
      if(!user || !user.isValidPassword(req.body.password) ) {
        throw new Error();
      }
      // generate JWT token and return it to the client
      var jwt = AuthService.generateJWT(user.toJSON());
      return res.json({
        message: "Successful login",
        token: jwt
      });
    })
    .catch(function(err) {
      return res.status(400).json({ msg: 'Authentication failed' });
    });
},
```

Koodikatkelma 5: Kirjautumisohjain [Github, 2016]

Koodikatkelmassa 5 on esimerkkisovelluksen kirjautumispäätepisteen ohjainfunktio, joka saa Sails.js-dokumentaation mukaisesti parametrikseen req- ja res-oliot. Funktiokutsua ennen tarkistetaan, että funktion suorittaminen ei vaadi kirjautumista. Funktion suorittaminen alkaa käyttäjän etsimisellä User-mallia hyödyntämällä tietokannasta body-parametrina annetun sähköpostiosoitteen avulla. Jos käyttäjää ei löydy, käyttäjän antamasta salasanasta muodostettu tiiviste ei vastaa tietokantaan tallennettua tiivistettä tai tietokantahaku muusta syystä palauttaa virheen, palautetaan asiakkaalle statuskoodi 400 sekä JSON-formaatissa viesti kirjautumisen epäonnistumisesta. Jos tunnisteet vastaavat tietokantaan tallennettua käyttäjää, luodaan käyttäjän tiedoista, poissulkien salasanasta luotu tiiviste, AuthService-palveluluokan avulla JSON Web Token -suojaustunnus, joka palautetaan käyttäjälle myöhempien kutsujen autentikointia varten. [Github, 2016]

Sails.js ei oletuksena sisällä testikirjastoa, mutta dokumentaatiossa on ohjeet yksinkertaisen testiympäristön pystyttämiseen [Sails.js, 2016c]. Demonstraatiosovelluksessa vain pääosassa olevan palvelinpään testaaminen on mahdollista. Testaaminen on toteutettu kutakin sovelluksen päätepistettä testaavana integraatiotestinä Mocha-, SuperTest- ja Chai-kirjastojen avulla. Mocha on JavaScriptille tarkoitettu testiohjelmistokehys, joka on suunniteltu erityisesti asynkronisten funktioiden testaamiseen ja joka tarjoaa selkeän ja jäsennellyn tavan kirjoittaa testejä [Mocha, 2016]. SuperTest on HTTP-pyyntöjen testaamiseen tarkoitettu kirjasto, jonka avulla HTTP-pyyntöjen tekeminen ja vastausten käsitteleminen on kattavasti ja yksinkertaisesti toteutettu [SuperTest, 2016]. Chai on väitteiden (assertions) käsittelyyn tarkoitettu kirjasto, jonka avulla voidaan varmistaa JavaScript-lauseiden ja -muuttujien totuusarvo ja joka palauttaa selkeästi muotoillun virheen, jos lause on loogisesti epätosi [Chai, 2016]. Esimerkkisovelluksen testit ajetaan backend-kansiosta komennolla ”npm test”.

Koodikatkelmassa 6 on katkelma autentikaatiopäätepisteen testistä, joka sijaitsee backend/test/integration/controllers/AuthController.test.js-tiedostossa. Testin alussa kerrotaan Mochan syntaksin mukaisesti describe-funktion avulla, että kyse on AuthController-luokan testaamisesta. Ennen kunkin testin suorittamista ajetaan beforeEach-fuktion avulla takaisinkutsuna annettu funktio, jossa luodaan HTTP-pyyntöjä lähettävä ja käsittelevä SuperTest-agentti sekä tallennetaan tietokantaan testikäyttäjiä usersMock-apuluokan avulla. Toinen koodikatkelman describe-funktio on ensimmäinen testi. Testissä lähetetään SuperTest-agentin avulla POST-typpinen HTTP-kutsu /api/auth/login-URL:iin. Kutsun body-osuudessa lähetetään tunnukset, jotka eivät vastaa mitään tietokantaan tallennettua käyttäjää. Kutsuketjun päätteeksi annetaan end-funktio ja sille takaisinkutsufunktio, joka saa parametrikseen mahdollisen virheen sekä pyynnön tuottaman vastauksen. Takaisinkutsufunktiossa tarkastetaan Chai-kirjaston expect-funktion avulla, että invalideilla tunnuksilla lähetettyyn kirjautumiskutsuun vastataan statuskoodilla 400. [Github, 2016]

```
describe('AuthController', function() {
  var users;
  var agent;

  beforeEach(function(done){
    agent = request.agent(sails.hooks.http.app);
    return usersMock.createUsers()
      .then(function(newUsers){
        users = newUsers;
        done();
      });
  });

  describe('#login()', function() {
    it('should not let an user to log in with invalid credentials', function (done) {
      agent
        .post('/api/auth/login')
        .send({ email: 'invalid', password: 'invalid'})
        .end(function(err, res){
          expect(res.statusCode).to.equal(400);
          done();
        });
    });
```

Koodikatkelma 6: Autentikaatiopäätepisteiden testaus [Github, 2016]

5.5. Demonstraatiosovelluksen toiminnollisuus

Esimerkkisovelluksen tarkoituksena on demonstroida kurssi-ilmoittautumista. Pieni osa toiminnollisuuksista on avoimia kaikille käyttäjille. Kirjautumaton käyttäjä voi kirjautua, rekisteröityä sovelluksen käyttäjäksi sekä listata tarjolla olevien kurssien nimet ja kuvaukset. Oppilaaksi (student) rekisteröitynyt käyttäjä pääsee näkemään tarkemmin kurssin tietoja, kuten luennoitsijan ja muiden kurssille osallistuvien käyttäjien nimet sekä ilmoittautumaan kurssille. Luennoitsijaksi (lecturer) kirjattu pystyy normaalin käyttäjän toiminnollisuuksien lisäksi lisäämään ja poistamaan uusia kursseja sekä poistamaan kurssilta luennoitsijan sekä ilmoittautumaan kurssille luennoitsijaksi. Järjestelmänvalvoja (admin) pystyy luennoitsijaroolin toiminnollisuuksien lisäksi asettamaan muita kirjautuneita käyttäjiä luennoitsijan ja oppilaan rooliin. [Github, 2016]

Sovelluksen asiakaspuolen näkymiä ovat etusivu, kirjautuminen, rekisteröityminen, kurssilista, kurssin yksityiskohdat, käyttäjäprofiili sekä käyttäjälista. Näkymien sisältö riippuu kirjautuneen käyttäjän roolista ja osa näkymistä on rajattu Angular 2:n @CanActivate-annotaation avulla näkymään ylipäätään vain tietyille käyttäjärooleille. Esimerkiksi profiilisivu aukeaa vain kirjatuneelle käyttäjälle ja käyttäjälista vain järjestelmänvalvojalle. Kurssilistan ulkonäkö taas on avoin kaikille käyttäjille, mutta kirjatumaton käyttäjä näkee vain kurssien nimet, oppilas lisäksi kunkin kurssin Open details -linkin ja luennoitsija sen lisäksi uuden kurssin luontiosuuden sekä kurssin poistamiseen tarkoitetun Delete-painikkeen. [Github, 2016]

6. REST ja turvallisuus

Fielding [2000] ei ota suoraan kantaa yksittäisiin turvallisuusuhkiin, mutta toteaa REST-rajapinnan helpottavan turvallisen verkkosovelluksen suunnittelua ja toteutusta. REST-rajapinnan määrittelemä yhdenmukaisuus helpottaa kehittäjän työtä turvallisuusuhkien etsimisessä ja paikkaamisessa, vaikka suoraa apua kaikkien uhkien ehkäisyyn ei pelkällä REST-pohjaisen arkkitehtuurin käyttämisellä olekaan. Kerroksista rakennettu järjestelmä myös suojaa ja abstrahoi resursseja niin, että niiden manipuloiminen on sallittua vain määritellyllä tavalla.

Fielding [2000] myöskään ei ota kantaa sovelluksen toteuttamistekniikoihin. REST-rajapinnan tarjoavan sovelluksen voi toteuttaa Javalla, PHP:llä tai esimerkiksi JavaScriptillä ja tietokanta, jossa dataa säilytetään voi olla MySQL, MariaDB tai NoSQL-kanta, kuten MongoDB. Koska osa turvallisuusuhkista on hyökkäyksiä tiettyä tekniikkaa vastaan, kuten esimerkiksi SQL-injektiot, ei REST-arkkitehtuurin valitseminen yksistään rajaa lähes-kään kaikkia uhkia pois. REST kuitenkin kannustaa järjestelmän rakentamiseen kerroksista, joista on pääsy vain lähimpiin kerroksiin.

Asiakassovellus ei siis REST-pohjaisessa järjestelmässä pääse suoraan lukemaan tietokantaa, vaan välissä oleva rajapinta voi esimerkiksi käyttää tietokannan manipulointiin turvalliseksi todettua komponenttia. Myöskään esimerkkisovelluksen käytössä olevaan MongoDB-tietokantaan ei voi tehdä suoraan kyselyitä, vaan kaikki käyttäjän ja tietokannan välinen vuorovaikutus tapahtuu REST-rajapinnan tarjoamien HTTP-metodien välityksellä.

Turvallisuutta suunniteltaessa valmiiden toteutuksien ja ohjelmistokehysten valitseminen on usein suositeltavaa, ja myös REST-pohjaista järjestelmää rakentaessa kannattaa hyödyntää mahdollisimman paljon valmiita, toimivaksi ja turvalliseksi todettuja ratkaisuita [OWASP, 2015]. Vaikka valittu ohjelmistokehys ei suoraan tukisikaan REST:iä ja esimerkiksi kehyksen sisäänrakennettu todentaminen hyödyntäisi REST:in vastaisesti istuntoja, saattaa kehykseen olla myös tarjolla vaihtoehtoisia autentikaatiokirjastoja.

Fielding [2000] kannustaa REST-rajapintaa kehittäessä haasteiden eriyttämiseen (separation of concerns). Ohjelma tulee siis rakentaa komponenteista, joilla on oma vastuualueensa ja jotka tarjoavat muille komponenteille yhdenmukaisen rajapinnan. Turvallisuuden kannalta haasteiden eriyttäminen on erityisen tärkeää, jolloin kunkin komponentin testaaminen onnistuu helposti esimerkiksi yksikkötesteinä ja jolloin yksittäisen komponentin muuttaminen tai jopa kokonaan vaihtaminen on mahdollisimman helppoa. Haasteiden eriyttäminen helpottaa OWASP:n [2013] luettelemista riskeistä erityisesti objektien ja funktioiden auktorisointia.

Esimerkkisovelluksessa kaikki toiminnollisuus on pyritty rakentamaan Sails.js- ja Angular 2 -dokumentaatioiden mukaisesti, jolloin kukin järjestelmän komponentti, kuten

ohjaimet, mallit, reititys ja auktorisointi, ovat vastuussa vain omasta erityisalastaan. Haasteiden eriyttämistä ei rajattu pelkästään palvelinpuolelle, vaan myös selainpuolen sovellus on pyritty rakentamaan mahdollisimman modulaariseksi. [Github, 2016]

REST auttaa turvaamaan funktiot auktorisoimattomilta käyttäjiltä kohtuullisen vaivattomasti, helposti testattavasti ja luotettavasti. Usein kukin rajapinnan tarjoama päätepiste (end point), eli HTTP-metodin ja URL:n yhdistelmä, voidaan eksplisiittisesti määritellä rajapinnan tarjoamaksi, jolloin virheellinen kutsu palauttaa esimerkiksi automaattisesti statuskoodin 404 (Not found) [OWASP, 2015]. Kukin rajapinnan päätepiste voidaan esimerkiksi RBAC-tyylisesti (Role Based Access Control) määritellä sallituksi tietylle roolille tai ACL-tyylisesti (Access Control List) kullekin käyttäjälle. Kun päätepisteet ovat omassa komponentissaan ja päätepisteiden auktorisointi omassaan, voidaan molempia hallita ja testata kohtuullisella vaivalla.

Objektien auktorisointi voidaan REST-pohjaisessa sovelluksessa toteuttaa esimerkiksi niin, että kuhunkin resurssiin on liitetty niiden käyttäjien tunnisteet, jotka voivat objektia muokata tai lukea. Yksittäisen objektin auktorisointi voidaan toteuttaa samaan tapaan kuin funktioiden auktorisointi eli erillisellä auktorisointikomponentilla. Auktorisointia vaativan objektin salliminen sitä pyytävälle käyttäjälle tarkistetaan siis ennen kuin vastausta aletaan muodostaa.

Esimerkkisovelluksessa auktorisointi perustuu sekä rooleihin että käyttäjän yksilölliseen tunnisteeseen. Kukin sovelluksen palvelinpään päätepiste auktorisoidaan erikseen ja kuhunkin päätepisteeseen voidaan liittää yksi tai useampi erilainen tarkistuskäytäntö, joita esimerkkisovelluksessa ovat muun muassa isAuthenticated (onko käyttäjä kirjautunut), isAdmin (onko kirjautuneen käyttäjän rooli järjestelmänvalvoja) ja isLecturer (onko kirjautuneen käyttäjän rooli luennoitsija). Toisaalta järjestelmänvalvojan lisäksi ainoastaan käyttäjä itse voi rekisteröidä itsensä kurssille sisään tai kurssilta ulos, mikä on varmistettu koodikatkelmassa 6 esitetyn isMy-käytänteen avulla. Käytänteessä siis varmistetaan, että URL:n sisältämä käyttäjätunniste on sama kuin sisään kirjautuneella käyttäjällä. [Github, 2016]

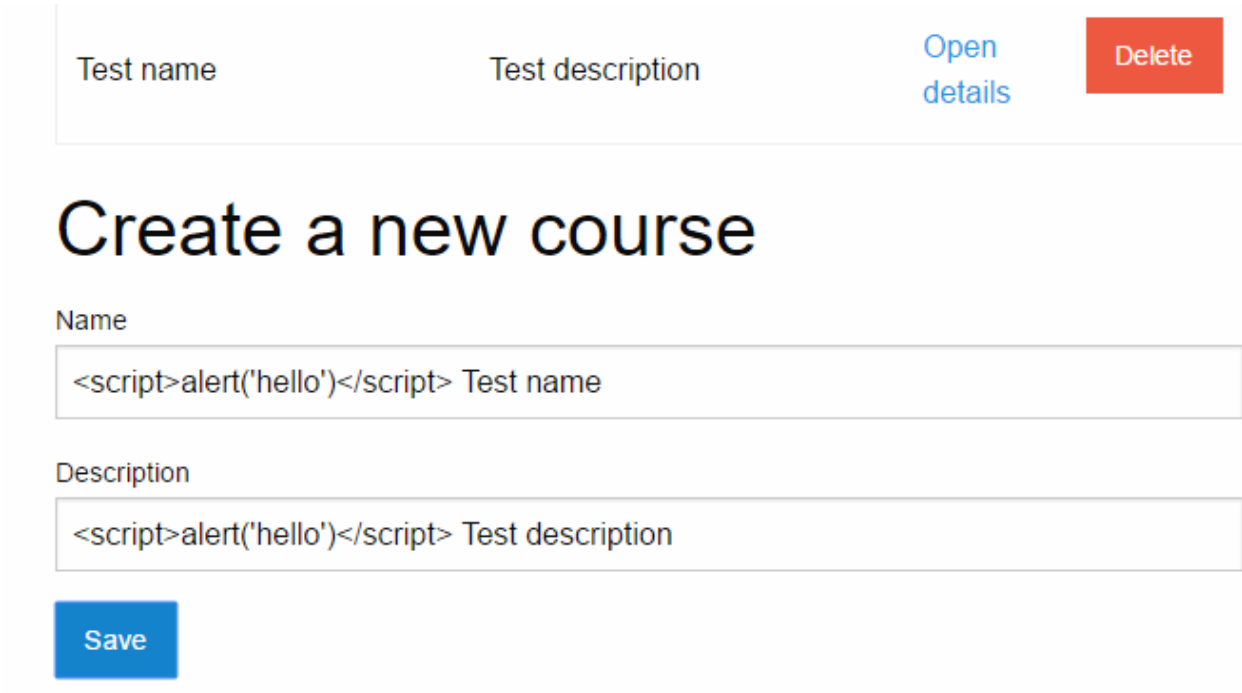
```
module.exports = function(req, res, next) {
  if (
    req.user.role === 'Admin' ||
    req.user.id.toString() === req.params.studentId.toString()
  ) {
    return next();
  }
  return res.forbidden('You are not permitted to perform this action.');
};
```

Koodikatkelma 7: isMy-käytäntö [Github, 2016]

REST-rajapinta on altis XSS-hyökkäyksille siinä missä muutkin internet-selaimilla käytettävät verkkopalvelut [OWASP, 2015]. XSS:n varalta kaikki käyttäjien tuottamat tieto-

kantaan tallennettavat tai muuten muille käyttäjille näytettävät syötteet tulee siis suodattaa tai muulla tavalla käsitellä niin, että selain ei missään tapauksessa suorita minkäänlaista toisen käyttäjän tuottamaa ohjelmakoodia syötettä esitettäessä [OWASP, 2015].

Esimerkkisovelluksessa kaikki syötteet puhdistetaan sanitize-html-kirjaston avulla [Github, 2016]. Se poistaa syötteestä esimerkiksi kaikki script-, style- ja textarea-tunnisteista sisällön tyystin sekä muista ei-halutuista tunnisteista vain itse tunnisteet [P'unk Avenue, 2016]. Kuvassa 12 käyttäjä on yrittänyt XXS-hyökkäystä lisäämällä kurssin nimeen ja kuvaukseen JavaScript-koodia. Lomakkeen yläpuolella olevasta uudesta kurssista voidaan kuitenkin nähdä, että sanitize-html-kirjasto on poistanut kaiken suoritettavan JavaScript-koodin molemmista syötteistä ja jättänyt jäljelle vain tunnisteiden ulkopuolisen tekstin.



Kuva 12: Esimerkki XXS-hyökkäysyrityksestä [Github, 2016]

OWASP [2015] suosittelee käsittelemään käyttäjän syötteet REST-rajapintaa hyödyntävien sovellusten tapauksessa erityisen tarkasti, sillä suositusten mukaisesti rakennettu rajapinta noudattaa selkeää standardia, jota hyökkäykseen suunniteltu ohjelmisto saattaa yrittää kuormittaa äärimmäisellä nopeudella ja toistojen lukumäärällä. Turvallisinta on avustaa käyttäjää antamaan vain tietynlaisia syötteitä. Postinumerokenttä voi esimerkiksi hyväksyä vain sellaisia postinumeroita, joita aiemmin syötetty kaupunki pitää sisällään. Jos syötettävä tieto on esimerkiksi vapaamuotoinen tekstikenttä, tulee syöte suodattaa. Jos syötteen suodattaminenkaan ei ole mahdollista, tulee tietoa esitettäessä varmistaa, että vaikka syöte sisältäisi ohjelmakoodia, sitä ei missään tapauksessa suoriteta.

Rajapintaa voidaan suojata myös vahvan muuttuja- ja sisältötyypityksen avulla. Jos syötteen tiedetään sisältävän esimerkiksi vain numeroita tai totuusarvoja, sallitaan syötteeksi vain sellaisiksi muunnettavat arvot. Palvelinsovellus voi myös varmistaa, että sille lähetetty sisältö on siinä muodossa, joka pyynnön otsakkeessa on annettu. Jos muoto on jotain muuta kuin mitä otsakkeessa on annettu ymmärtää, tulee pyyntö hylätä. [OWASP, 2015]

Esimerkkisovelluksessa oppilaaksi rekisteröityneen käyttäjän syöttämä tieto on minimoitu. Kirjautumisen ja rekisteröitymisen lisäksi käyttäjä ei käytännössä voi olla muussa vuorovaikutuksessa sovelluksen kanssa, kuin rekisteröitymään ulos tai sisään kurssille. XXS-hyökkäyksien estämiseksi toteutettu sisällön puhdistaminen varmistaa, että käyttäjä ei pysty aiheuttamaan OWASP:n [2013] esittelemiä validoimattomia uudelleenohjauksia eikä XXS-hyökkäyksiä. [Github, 2016]

Kuten tutkielmassa aiemmin todettiin, toteutetaan cross site request forgery -suojaus usein palvelimella generoitujen suojaustunnusten avulla. Jokainen asiakkaan tekemä ei-turvallinen kutsu, eli POST-, PUT- tai DELETE-metodin avulla toteutettu pyyntö, tulee suojata CSRF-hyökkäykseltä niin, että kutsu hyväksytään vain, jos se sisältää validin tokenin esimerkiksi kyselyn body-osassa tai otsakkeessa [OWASP, 2016]. Token generoidaan ja validoidaan normaalisti istunnon perusteella, mikä on puhtaan REST:n kannalta ongelmallista, sillä REST-rajapinnan tulisi olla tilaton.

CSRF-hyökkäys on kuitenkin mahdollinen vain, jos selain itsessään liittää automaattisesti käyttäjän tunnistetietoja kutsuun. Selaimen automatiikkaa hyödyntävät esimerkiksi HTTP-todentaminen sekä istuntoihin perustuva todentaminen. Jos todentaminen on sen sijaan toteutettu siten, että automaattisesta autentikaatiosta vastaa selaimen sijaan asiakassovellus ja että käyttäjän tunnistamiseen tarvittava tieto on selaimen evästeiden sijasta tallennettu esimerkiksi selaimen paikalliseen tietovarastoon, jonka sisältöä selain ei koskaan liitä automaattisesti kutsuihin, on CSRF-hyökkäys tehoton. CSRF-suojaus ei siis ole välttämätön, jos todentaminen perustuu esimerkiksi suojaustunnuksiin, kuten tutkielmassa aiemmin esiteltyjen JWT-perustaisen todentamisen tai OpenID Connectin tapauksessa. [OWASP, 2016]

Esimerkkisovelluksessa käyttäjän autentikoimiseen käytettyä suojaustunnusta ei lisätä evästeisiin, vaan se tallennetaan selaimen paikalliseen tietovarastoon, josta asiakassovellus lisää tunnuksen kunkin HTTP-kutsun Authorization-otsakkeeseen. Tunnukselle on määritetty myös kahden tunnin voimassaoloaika, jonka jälkeen käyttäjältä vaaditaan uudelleen sähköpostiosoite ja salasana, ennen kuin autentikointia vaativien funktioiden kutsuminen onnistuu. Vaikka hyökkääjä onnistuisikin tekemään palvelinpuolen REST-rajapinnalle kutsuja käyttäjän selaimesta, kutsut eivät sisältäisi käyttäjän todentamiseksi vaadittua suojaustunnusta. CSRF-hyökkäys ei siis aiheuttaisi esimerkkisovelluksen tapauksessa käyttäjälle tai sovellukselle haittaa. [Github, 2016]

Myös REST-rajapinta tulee suunnitella niin, että arkaluontoista tietoa ei paljasteta sellaisille osapuolille, jota voisivat vahingollisesti hyötyä siitä. Asiakkaan ja palvelimen väliset transaktiot on syytä salata esimerkiksi SSL/TLS-yhteyden avulla, jos transaktioissa kulkee mitä tahansa arkaluontoista tietoa. Tietokanta tulee suojata niin, että arkaluontoinen tieto esimerkiksi salataan tai hajautetaan ja että tietokantayhteyden voi muodostaa vain luotettava taho. REST-rajapintaa hyödyntävän asiakassovelluksen tulee käsitellä kaikkea lokaalisti tallennettavaa tietoa niin, että sisältöön pääsee käsiksi vain luotettava taho.

Esimerkkisovellus käyttää samalla palvelimella toimivaa MongoDB-tietokantaa tiedon säilyttämiseen. Salasana on käytännössä ainoa arkaluontoinen tieto, jota tietokantaan tallennetaan ja se on suojattu tiivistämällä se käyttämällä bcrypt-nodejs-kirjastoa, joka salaa merkkijonon Blowfish-algorimin avulla [Isaacson et al., 2015]. Sovelluksen käyttämä tietokanta on tuotantokäytössä tarkoitus konfiguroida niin, että siihen ei pääse käsiksi muuten kuin samalta palvelimelta. Tarvittaessa tietokantakonfiguraatiot on kuitenkin helposti muutettavissa ja sovellus voi käyttää missä tahansa sijaitsevaa HTTP-yhteyden yli käytettävää MongoDB-kantaa toimiakseen, jolloin myös useampi palvelininstassi voi hyödyntää samaa tietokantaa. [Github, 2016]

Sails.js:n tapauksessa SSL-sertifikaattien hyödyntäminen transaktioiden salaamiseen on selkeästi dokumentoitu [Sails.js, 2016a], joten myös esimerkkisovelluksen tuotantokäyttöä varten SSL-sertifikaattien hankkiminen ja niiden tallentaminen sovelluksen konfiguraatioihin http-yhteyttä varten olisi erittäin suotavaa, jolloin mies välissä -hyökkäyksen tekeminen sovellusta vastaan olisi käytännössä hyödytöntä vahvan salauksen ansiosta.

7. REST ja todentaminen

REST-rajapinnan kanssa käytettäviä autentikaatiokeinoja rajaavat REST:n rajoitukset ja erityisesti vaatimus REST-rajapintaa tarjoavan palvelimen tilattomuudesta, sillä moni autentikaatiokeino vaatii toimiakseen istuntojen käyttämisen. Vaikka istuntojen hyödyntäminen onkin yhtä mutkatonta toteuttaa REST-rajapinnan kuin minkä tahansa muun webpalvelun kanssa ja esimerkiksi OWASP [2015] suosittelee käyttämään istuntoihin perustuvaa autentikaatiota, kehottaa Fielding [2000] välttämään istuntojen käyttämistä sekä palvelimelle että evästeisiin tallennettuna.

7.1. REST ja HTTP-todentaminen

Sekä Digest- että Basic access -todentaminen on suunniteltu perinteisen tietokoneselaimen kanssa toimiviksi standardeiksi autentikoida käyttäjä, jolloin kehittäjä voi hyödyntää valmiita toteutuksia ja jolloin laskenta jää palvelimen ja selaimen vastuulle. Palvelin palauttaa todentamista vaativia resursseja kyseltäessä haasteen, jos sellaiseen ei löydy vastausta kyselyn otsakkeista. Selain pyytää tunnuksia käyttäjältä oman kirjautumisikkunansa avulla kerran, minkä jälkeen se tallentaa tunnukset välimuistiinsa ja palauttaa lopulta vastauksen palvelimen antamaan haasteeseen haasteessa pyydetyllä tavalla.

HTTP-todentaminen on REST-sovellusten kannalta haastava todentamistapa. Basic access -todentamisessa haasteena on erityisesti tietoturva, sillä pelkkä standardissa kuvattu Base64-koodaus ei suojaa tunnuksia hyökkäyksiltä. HTTPS-yhteyden kanssa toteutettuna Basic access on mies välissä -hyökkäyksen suhteen turvallisempi keino, mutta ongelmaksi jää edelleen käyttäjän uloskirjaaminen, sillä tunnukset ovat voimassa juuri niin kauan kuin ne erikseen vaihdetaan. Jos hyökkääjä siis pääsee selaimen lukemaan välimuistiin tallennetut tunnukset, ne ovat ilman erillisiä toimenpiteitä rajattomasti voimassa.

Digest-todentaminen on selvästi HTTP-todentamistavoista turvallisempi erityisesti MD5-salauksen ansiosta, jolloin edes HTTPS-yhteyttä ei välttämättä tarvita. Digest-todentamisessa myös tunnusten vanheneminen on mahdollista toteuttaa standardin kuvaamalla tavalla, joten kyselyotsakkeiden joutuminen vääriin käsiin on konkreettinen uhka vain vanhenemisajan sisällä.

Digest-todentamisessa ongelma jääkin enimmäkseen asiakaskomponenttien toteuttajan vastuulle, sillä REST-rajapinta ei ota kantaa siihen, mikä selain tai laite ylipäätään pyyntöjä lopulta tekee. Yleisimmät selaimet, kuten Google Chrome, Mozilla Firefox sekä Internet Explorer osaavat käsitellä Digest-todentamiseen liittyvän haasteen oikein, esittää käyttäjälle halutunlaisen kirjautumisikkunan, muodostaa otsakkeen Digest-todentamisen standardin vaatimalla tavalla sekä liittää otsakkeen kaikkiin selainistunnon sisällä tapahtuviin kutsuihin. Jos kysely tehdäänkin selaimesta, johon ei ole toteutettu tukea Digest-todentamiselle tai esimerkiksi puhelimeen tai älykelloon toteutetulla applikaatiolla, jää

palvelimen palauttaman haasteen tulkitseminen kirjautumisikkunan toteuttaminen sekä monimutkaisen otsakkeen muodostaminen asiakassovelluksen kehittäjä vastuulle.

Yleisimpiä selaimia käyttäessä ongelmaksi voidaan HTTP-salauksen suhteen nähdä myös käyttäjäkokemus, sillä kehittäjä ei voi vaikuttaa haasteeseen vastaavan kirjautumisikkunan ulkonäköön. Ongelmistaan huolimatta HTTP-todentaminen on palvelinpuolelle yleensä yksinkertainen toteuttaa, sillä useille eri WWW-tekniikoille löytyy omat HTTP-todentamislisäosansa. HTTP-todentaminen on myös täysin tilaton todentamisratkaisu ja näin ollen toimii REST-rajapinnan määritelmän mukaisesti. Palvelimen ei siis tarvitse pitää yllä minkäänlaista istuntoon tai muuhun ratkaisuun riippuvaa tilaa, joka yhdistäisi kutsut tai ratkaisisi käyttäjän todentamiseen liittyviä ongelmia, vaan kaikki autentikaatioon vaadittava tieto kulkee kutsujen Authorization-otsakkeessa.

7.2. REST ja istuntoihin perustuva todentaminen

Istuntoihin perustuva todentaminen on muun muassa OWASP:n [2015] suosittelema tapa autentikoida käyttäjä REST-pohjaisissa sovelluksissa. Jos istunto tallennetaan palvelimen puolelle esimerkiksi tietokantaan tai palvelimen omaan muistiin, ei istuntoihin perustuva autentikaatio kuitenkaan täytä Fieldingin [2000] esittämää vaatimusta palvelimen tilattomuudesta, eikä istuntoja hyödyntävää rajapintaa voida siis pitää Fieldingin määrittelemänä REST-rajapintana.

Palvelimelle tallennettujen istuntojen ylläpitäminen hankaloittaa erityisesti järjestelmän palvelinpuolen hajauttamista sekä aiheuttaa ylimääräistä kuormitusta sekä tietokantapyyntöjen että fyysisen tilan suhteen palvelimelle. Jos istunto tallennetaan palvelimen omaan välimuistiin, tulee kyseisen asiakkaan kaikki pyynnöt ohjata aina kyseiselle palvelininstanssille, mikä on ongelmallista, sillä palvelimen kuormitusta on hankala ennustaa. Palvelimen välimuistiin tallennettu istunto on hankala myös siksi, että palvelininstanssin kaatuminen aiheuttaa myös istunnon tuhoutumisen, jolloin uusi todentaminen on välttämätöntä.

Evästeisiin tallennettu istunto sen sijaan kuormittaa verkkoyhteyttä ja on ilman asianmukaista salausta mies välissä -hyökkäykselle altis. Evästeeseen tallennettu istunto tulisi siis salata ja toteuttaa niin, että istunnolle annetaan sitä luotaessa voimassaoloaika, jonka jälkeen autentikointi tulisi uusia käyttäjätunnuksen ja salasanan avulla.

Huolellisesti suunniteltuna ja toteutettuna istuntoihin perustuva autentikaatio on kuitenkin turvallinen, eri asiakaslaitteiden- ja sovellusten kannalta dynaaminen ja valmiiden lisäosien ansiosta kevyesti toteutettavissa oleva ratkaisu, joka mahdollistaa myös käyttäjän uloskirjaamisen sovelluksesta. Jos käyttäjädataa ylläpidetään palvelimen päässä ja selain liittää kutsuihin vain istunnon tunnisteen, on istuntoihin perustuva todentaminen myös verkkoyhteyden kuormituksen kannalta kevyt ratkaisu.

7.3. REST ja OpenID-todentaminen

OpenID keventää erityisesti REST-rajapintaa tarjoavan sovelluksen turvallisuustaakkaa, sillä kriittisin osuus todentamisesta on delegoitu asiakkaan ja todentamispalveluntarjojan välille. OpenID myös vapauttaa kehittäjän käyttäjätunnuksen ylläpitämisestä, kuten tunnusten luomisesta, poistamisesta ja salasanan vaihtamisesta.

Vaikka OpenID onkin kehittäjän kannalta vaivaton tapa toteuttaa käyttäjän tunnistaminen, sen käyttäminen vaatii kirjautumistiedon ylläpitämistä palvelimen puolelle luodussa istunnossa, mikä rikkoo REST:n tilattomuuden vaatimuksen. Ilman salattua HTTPS-yhteyttä OpenID:tä ei voida pitää muutenkaan turvallisena autentikaatiotapana, sillä se on altis mies välissä -hyökkäyksille. OpenID on myös käyttäjän yksityisyyden kannalta kyseenalainen vaihtoehto, koska se antaa identiteetin tarjoajalle mahdollisuuden käyttäjän seuraamiseen.

7.4. REST ja JWT-perustainen todentaminen

Koska JWT-perustainen todentaminen ei vaadi evästeitä tai istuntoja onnistuakseen, sitä voidaan pitää erityisen yhteensopivana REST-rajapinnan kanssa. JWT-avainta luotaessa myös avaimen voimassaoloaika on helposti asetettavissa, joten vaikka avain joutuisi väärin käsiin, olisi se voimassa vain ennalta määritettyyn voimassaoloaikaan asti. JWT on pienen tietosisällön kanssa toteutettuna kevyt lähettää asiakassovelluksesta palvelimelle, joten se ei juurikaan kuormita asiakkaan ja palvelimen välistä verkkoyhteyttä. JWT-perustainen autentikaatio on myös kevyt toteuttaa, sillä valmiita ja usein myös suosittuja ja siten myös todennäköisesti laajasti vertaisarvioituja JWT-kirjastoja löytyy kaikille suosituimmille verkkosovellusten toteuttamiseen erikoistuneille tekniikoille kuten Javalle, Node.js:lle, PHP:lle, Rubylle sekä Pythonille. [JWT, 2015]

JWT-perustainen todentaminen on altis mies välissä -hyökkäykselle. Riskialttein vaihe JWT-todentamisessa on käyttäjän tunnusten lähettäminen palvelimelle kirjautumisen yhteydessä, jolloin ilman asianmukaista salausta hyökkääjä pääsee tunnuksiin käsiksi ja saa tunnusten vaihtamiseen asti rajattoman pääsyn uhrin käyttäjätilille. Myöhemmissä kutsuissa, joissa todentaminen tapahtuu JWT:n avulla, riski on merkittävästi pienempi, jos tunnukselle on asetettu mahdollisimman lyhyt voimassaoloaika ja jos tunnuksen tietosisältöön ei ole tallennettu minkäänlaista arkaluontoista dataa. Mies välissä -hyökkäyksien varalta yhteys kannattaa suojata SSL/TLS-salauksella.

Esimerkkisovelluksen käyttäjätunnistaminen on toteutettu JWT-perustaisena todentamisena. Käyttäjä kirjoittaa kirjautumislomakkeelle sähköpostiosoitteen ja salasanan, jotka asiakassovellus lähettää REST-rajapinnalle. Palvelin verifioi tunnukset ja palauttaa oikeita tunnuksia vastaan allekirjoitetun JWT:n joka sisältää käyttäjän User-taulun sisällön ilman tiivistettyä salasanaa. Asiakassovellus tallentaa palvelimen palauttaman JWT-suojaustunnuksen selaimen paikalliseen tietovarastoon (local storage). Kaikki asiakassovelluksen tekemät HTTP-pyynnöt tehdään HttpService-luokan avulla, joka lukee selaimen

paikallisesta tietovarastosta session-token-nimellä tallennetun JWT-suojaustunnuksen ja liittää tunnuksen kaikkien HTTP-pyyntöjen Authorization-otsakkeeseen muodossa ”JWT <suojaustunnus>”. [Github, 2016]

Palvelinsovellus autentikoi kunkin autentikoitavaksi määritetyn pyynnön koodikatkelmassa 8 esitetyn isAuthenticated-käytännön avulla, joka lukee ja verifioi Authorization-otsakkeesta JWT-avainsanan jälkeen löytyvän tunnuksen. Jos tunnuksen verifioiminen onnistuu, etsitään tietokannasta käyttäjä tunnisteella, joka on tallennettu JWT-suojaustunnuksen tietosisältöön. Jos käyttäjä löytyy, tallennetaan löytyneen käyttäjän tiedot req-olioon myöhempää käyttöä, kuten auktorisoimista, varten. [Github, 2016]

```
try {
  /**
   * Look Authorization header for JSON web token
   * Required format is Authorization = "JWT <JSON Web Token>"
   */
  var authHeader = req.headers['authorization'];
  authHeader = authHeader.split(" ");

  if(authHeader[0] === "JWT"){
    var jwt = authHeader[1];
    // verifyJWT() throws an error, if token is not valid
    var decoded = AuthService.verifyJWT(jwt);
    return User.findOne({id: decoded.id})
      .then(function(authUser){
        if(authUser) {
          req.user = authUser;
          next();
          return null;
        }
      });
  }
} catch(err) {
  return res.forbidden('You are not permitted to perform this action.');
}
return res.forbidden('You are not permitted to perform this action.');
```

Koodikatkelma 8: isAuthenticated-käytäntö [Github, 2016]

Toteutus ei yksinään ole turvallinen, vaan mahdollistaa esimerkiksi mies välissä -hyökkäyksen kahden tunnin sisällä kirjautumisesta, jonka jälkeen JWT on määritetty vanhenemaan. Merkittävästi turvallisemmaksi toteutus sen sijaan tulee, jos sovelluksen kanssa otetaan käyttöön SSL-sertifikaatti Sails.js-projektin anatomiaosuudessa esitellyn config/local.js-tiedoston dokumentaation mukaisesti [Sails.js, 2016a].

7.5. REST ja OpenID Connect -todentaminen

Kuten Li ja Mitchell [2015] osoittavat, ei OAuth 2.0 -standardiin perustuva OpenID Connect -todentaminen takaa automaattisesti turvallista tapaa autentikoida käyttäjä, mutta standardia ja palveluntarjoajan omaa dokumentaatiota noudattamalla sekä salattua yh-

teyttä käyttämällä autentikaatiosta voidaan kehittää turvallinen ja helposti ylläpidettävä kokonaisuus. Toteutus on myös käyttäjän kannalta kevyt, sillä hänen ei välttämättä tarvitse luoda uusia tunnuksia rekisteröityäkseen uuden sovelluksen käyttäjäksi.

Koska OpenID Connect hyödyntää JSON Web Token -suojaustunnuksia, se ei pakota käyttämään palvelimelle tai evästeisiin tallennettuja istuntoja ja on siten erityisesti palvelimen tilattomuuden kannalta REST-yhteensopiva tapa todentaa käyttäjä. JWT:n ansiosta OpenID Connect -istunto voidaan asettaa vanhenemaan määrätyn ajan kuluessa ja asiakassovellukselle voidaan palauttaa suojaustunnuksen mukana esimerkiksi käyttäjäkohtaista dataa.

Samoin kuin OpenID 2.0, myös OpenID Connect mahdollistaa todentamispalveluiden delegoimisen kolmannelle osapuolelle ja vapauttaa näin riippuvan osapuolen kehittäjän resursseja erityisesti käyttäjätunnusten hallinnoinnin suhteen. Myös OpenID Connectia hyödynnettäessä täytyy kuitenkin muistaa, että todennuksen tarjoaja saattaa rekisteröidä jokaisen uuden istunnon aloittavan autentikoinnin ja näin kolmas osapuoli voi helposti seurata käyttäjän verkkokäyttäytymistä.

7.6. REST ja Tupas-tunnistaminen

Tupas-tunnistaminen on vahva tunnistaminen, jonka turvallisuus on tunnistusta tarjoavan pankin ja palveluntarjoajan välillä tarkasti määritelty kokonaisuus. Tupas-tunnistaminen soveltuu kaikille yksityisille ja julkisoikeudellisille palveluntarjoajille, jotka harjoittavat Suomen lainsäädännössä määriteltyä sähköistä toimintaa, joten Tupas-tunnistamista voi hyödyntää lainsäädännön puitteissa aina, kun käyttäjän tunnistaminen on kriittistä. [Finanssialan keskusliitto, 2013b]

Samoin kuin OpenID:n ja OpenID Connectin tapauksissa, myös Tupas-tunnistaminen vapauttaa käyttäjän ja kehittäjän ylimääräisten kirjautumistunnusten ylläpidosta. Toisaalta Tupas-tunnistamisen käyttöönotto vaatii raskaan prosessin, joka sisältää muun muassa sopimusten kirjoittamisen sekä teknisen toteutuksen suunnittelun erikseen kunkin Tupas-tunnistamista tarjoavan pankin kanssa, joita palveluntarjoaja haluaa hyödyntää. Pankki voi myös halutessaan kieltäytyä toimittamasta Tupas-tunnistamista palveluntarjoajalle, jos palveluntarjoaja toimii pankin mielestä hyvän tavan tai lain vastaisesti. [Finanssialan keskusliitto, 2013b]

Tupas-tunnistamiseen kuuluvaa pankin palveluntarjoajan välisiä transaktioita ei voi toteuttaa ilman salattua yhteyttä, joten sitä voidaan pitää turvallisena myös mies välissä -hyökkäyksiä sen avulla tehtäviä toistettavia kutsuja vastaan. Myöskään transaktioiden tietosisältöön ei voi tehdä hyökkäyksen mahdollistavia muutoksia, sillä tietosisältö allekirjoitetaan sekä transaktioissa, jotka lähtevät palveluntarjoajalta pankille että transaktioissa, jotka lähtevät pankilta palveluntarjoajalle. [Finanssialan keskusliitto, 2013b]

Koska Tupas-tunnistusta kehittävä ja ylläpitävä Finanssialan keskusliitto ei määrittele kuinka Tupas-tunnistamisen jälkeen käyttäjän palvelimelle lähettämät kutsut tulisi auten-

tikoida, jää kehittäjälle vapaat kädet myös tilattoman ja siten myös REST-yhteensopivan todentamisratkaisun luomiseen. Turvallinen ratkaisu hyödyntää kaikkien transaktioiden salaamiseen SSL-yhteyttä, jotta mies välissä -hyökkäyksen avulla ei voida toistaa käyttäjän lähettämiä kutsuja. Tilattomuuden takia ratkaisu ei voi perustua istuntoihin, joten esimerkiksi JWT-perustainen ratkaisu voisi olla käytännöllinen. JWT:tä muodostettaessa tietosisältöön ei kuitenkaan tule sisällyttää mitään arkaluontoista tietoa. Käyttäjän identifioivan subject-avaimen arvo voisi esimerkiksi olla esimerkiksi palveluntarjoajan oman tietokannan käyttäjätunniste, eikä esimerkiksi pankin palauttama tunniste, jonka avulla pankin tunnistama käyttäjä voidaan yhdistää tiettyyn palveluntarjoajan asiakkaaseen.

8. Yhteenveto

Representational State Transfer -arkkitehtuurin valinta ei yksistään riitä tietoturvallisen sovelluksen rakentamiseen, mutta se helpottaa turvallisen rajapinnan suunnittelua, kehittämistä, testaamista ja ylläpitoa. Vaikka Fielding ei väitöstutkimuksessaan ota REST-rajapinnan toteutuksen yksityiskohtiin kantaa, on REST:stä muodostunut siinä määrin käytetty arkkitehtuurityyli, että turvalliseksi todettuja käytäntöjä on kehitetty pitkälle ja niiden noudattaminen on mutkatonta.

REST kannustaa haasteiden eriyttämiseen ja järjestelmän modulaarisuuteen, mikä edesauttaa järjestelmän jatkokehitystä ja testausta ja siten myös turvallisuutta. REST ei ota kantaa toteuttamistekniikoihin, joten itse rajapinnan ja varsinkin sitä hyödyntävät asiakassovellukset voi toteuttaa lähes millä tahansa HTTP-yhteyttä tukevalla tekniikalla.

Vaikka erilaisia todentamistekniikoita on useita, ei mikään tavoista ole ongelmaton. Jos järjestelmän on tarkoitus noudattaa tiukasti Fieldingin [2000] määrittelemiä rajoituksia, on sekä kaikki sekä palvelinpäässä että evästeiden avulla toteutetut istuntoihin perustuvat autentikaatiotavat, kuten OpenID 2.0 ja perinteinen palvelimelle luotuun istuntoon perustuva autentikointi unohdettava.

Mies välissä -hyökkäys mahdollistaa autentikaatiotavasta riippumatta vähintäänkin kertaalleen toistettavan hyökkäyksen, joten erityisesti kriittisiä sovelluksia toteutettaessa SSL/TLS-salattu yhteys on välttämätön. Salaamattomalla yhteydellä turvallisuuden kannalta erityisen kriittinen on kuitenkin HTTP Basic access -todentaminen, jossa käyttäjätunnus ja salasana lähetetään palvelimelle salaamattomana. Json Web Tokeneihin perustuvassa todentamisessa tulee muistaa, että suojaustunnuksen tietosisältöön ei tule tallentaa sellaista tietoa, joka voisi olla vääriin käsiin joutuessa käyttäjän tai järjestelmän turvallisuuden kannata kriittistä.

Sekä kehittäjän että käyttäjän kannalta käyttäjätunnusten hallinnointi saattaa olla työlästä, joten autentikoinnin ulkoistaminen on houkutteleva ajatus. Kaikki kolmansia osapuolia hyödyntävät autentikointitavat, kuten OpenID ja OpenID Connect, pienentävät sekä kehittäjän että käyttäjän työmäärää, mutta antavat kolmansille osapuolille mahdollisuuden seurata käyttäjän verkkokäyttäytymistä. Oikoen ja erityisesti ilman HTTPS-yhteyttä toteutettuna ne myös aiheuttavat merkittäviä turvallisuusuhkia.

Viiteluettelo

Subbu Allamaraju. 2010. *RESTful Web Services Cookbook.* O'Reilly Media.

Mike Amundsen and Leonard Richardson. 2013. *RESTful Web APIs*. O'Reilly Media.

T. Berners-Lee, R. Fielding and H. Frystyk. 1996. *Hypertext Transfer Protocol -- HTTP/1.0*. Available: https://www.w3.org/Protocols/HTTP/1.0/spec.html. Checked 4.4.2016.

J. Bradley, M. Jones and N. Sakimura. 2014. *JSON Web Token (JWT)*. Available: http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-20. Checked 4.4.2016.

J. Bradley, M. Jones, B. de Meideros, C. Mortimore and N. Sakimura. 2013. *OpenID Connect Basic Client Implementer's Guide 1.0.* Available: *HTTP://openid.net/specs/openid-connect-basic-1_0-31.html.* Checked 4.4.2016.

Mike Cantelon, Marc Harter, T. J. Holowaychuk and Nathan Rajlich. 2014. *Node.js in Action*. Manning Publications Co.

Chai. 2016. *API Reference*. Available: http://chaijs.com/api/. Checked 4.4.2016.

Nigel Chapman and Jenny Chapman. 2012. *Authentication and Authorization on the Web.* MacAvon Media.

Yu Chin Cheng, Xiang-Wen Huang, Chin-Yun Hsieh and Cheng Hao Wu. 2015. A token-based user authentication mechanism for data exchange in RESTful API. In: *Proc. of the 18th International Conference on Network-Based Information Systems*, 601-606.

Michael Cross. 2007. *Developers Guide to Web Application Security*. O'Reilly Media.

Bart van Delft and Martijn Oostdijk. 2010. A Security Analysis of OpenID. In: *Proc. of the Second IFIP WG 11.6 Working Conference, IDMAN 2010*, 73-84.

Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph. D. Dissertation, Information and Computer Science, University of California, Irvine.

Finanssialan keskusliitto. 2013a. *Pankkien TUPAS-tunnistuspalvelu palveluntarjoajille. Palvelukuvaus ja palveluntarjoajan ohje. Versio 2.4.* Saatavilla: http://www.finanssiala.fi/maksujenvalitys/dokumentit/Tupas_varmennepalvelu_V_2.4.pdf#search=tupas. Tarkistettu 4.4.2016.

Finanssialan keskusliitto. 2013b. *Pankkien Tupas-tunnistuspalvelun tunnistusperiaatteet V2.0c.* Saatavilla: http://www.finanssiala.fi/maksujenvalitys/dokumentit/Tupas_tunnistusperiaatteet_v20c.pdf#search=tupas. Tarkistettu 4.4.2016.

Finlex. 2009. *Laki vahvasta sähköisestä tunnistamisesta ja sähköisistä allekirjoituksista*. Saatavilla: http://www.finlex.fi/fi/laki/ajantasa/2009/20090617. Tarkistettu 4.4.2016.

Minko Gechev. 2016. *angular2-seed*. Available: https://github.com/mgechev/angular2-seed. Checked 4.4.2016.

Github. 2016. *courseEnrollment*. Available: https://github.com/Sirenj/courseEnrollment. Checked: 4.4.2016.

Google. 2016. *Angular docs.* Available: https://angular.io/docs/ts/latest/. Checked: 4.4.2016.

Noah Isaacson, Alex Murray, Nicolas Pelletier and Josh Rogers. 2015. *bcrypt-nodejs*. Available: https://github.com/shaneGirish/bcrypt-nodejs. Checked 4.4.2016.

JWT. 2015. *Introduction to JSON Web Tokens*. Available: https://jwt.io/introduction/. Checked 4.4.2016.

Wanpeng Li and Chris J Mitchell. 2015. *Analysing the Security of Google's implementation of OpenID Connect*. University of London, Information Security Group.

Mark Massé. 2012. *REST API Design Rulebook*. O'Reilly Media.

Mocha. 2016. *mochajs.org*. Available: http://mochajs.org/. Checked 4.4.2016.

OWASP. 2013. *The Ten Most Critical Web Application Security Risks*. Available: http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf. Checked 4.4.2016.

OWASP. 2015. *REST Security Cheat Sheet*. Available: https://www.owasp.org/index.php/REST_Security_Cheat_Sheet. Checked 4.4.2016.

OWASP. 2016. *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*. Available: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet. Checked 4.4.2016.

P'unk Avenue. 2016. *sanitize-html*. Available: https://github.com/punkave/sanitize-html Checked 4.4.2016.

Justin Richer. 2014. *User Authentication with OAuth 2.0*. Available: http://oauth.net/articles/authentication/. Checked 4.4.2016.

Sails.js. 2016a. *Anatomy of a Sails App.* Available: http://sailsjs.org/documentation/anatomy/my-app/api. Checked: 4.4.2016.

Sails.js. 2016b. *Getting Started.* Available: http://sailsjs.org/get-started. Checked: 4.4.2016.

Sails.js. 2016c. *Sails.js Documentation: Core Concepts.* Available: http://sailsjs.org/documentation/concepts/routes Checked: 4.4.2016.

SuperTest. 2016. *SuperTest.* Available: https://github.com/visionmedia/supertest. Checked 4.4.2016.

John R. Vacca. 2009. *Computer and Information Security Handbook*. Morgan Kaufmann.