OAuth 2.0 ja Authorization code grant

Demosovellus: Spot-A-Hit

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Visamäki, tietojenkäsittely

Kevät, 2017

Niko Tastula

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittely
Visamäki

Tekijä Niko Tastula **Vuosi** 2017

Työn nimi OAuth 2.0 ja Authorization code grant

Työn ohjaaja/t Tommi Lahti

TIIVISTELMÄ

Tämän opinnäytetyön tarkoitus on tutustua OAuth 2.0 -autorisointiprotokollaan luomalla yksinkertainen web-sovellus nimeltään Spot-A-Hit, joka luodaan Javan Spring-sovelluskehyksen Spring Boot -lisäosan avulla. Sovelluksen ideana on valtuuttaa se käyttämään OAuth 2.0 -protokollan avulla sovelluksen käyttäjän omistamia tietoja, jotka sijaitsevat musiikkipalvelu Spotifyn palvelimella.

Teoriaosuus sisältää käytännönläheistä tietoa OAuth 2.0 -autorisointiprotokollasta sekä erilaisista mahdollisista autorisointimalleista joita OAuth 2.0 tarjoaa. Opinnäytetyössä keskitytään lähes pelkästään Authorization code grant -autorisointimalliin, joka on yleisesti suositelluin malli tietoturvansa takia. Opinnäytetyössä esitellään teoria tämän autorisointimallin eri vaiheista käyttäen Spotifyn tarjoamaa REST-rajapintaa. Lisäksi opinnäytetyöhön kuuluu pieni asiasanasto aiheeseen liittyvistä termeistä.

Käytännön osuus koostuu varsinaisen sovelluksen ominaisuuksiin tutustumisesta, sekä käytännön toteutuksesta autorisointimallin eri vaiheissa. Sovelluksessa käytetyt tekniikat sekä ohjelmakirjastot on eritelty omassa kappaleessaan, ja kaikista löytyy pieni kuvaus. Lisäksi esitellään ohjelmalistausta ohjelman käyttämistä luokista soveltuvin osin.

Avainsanat OAuth 2.0, autorisointi, Java

Sivut 27 sivua, joista liitteitä 0 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information technology
Visamäki

Author Niko Tastula **Year** 2017

Subject OAuth 2.0 and Authorization code grant

Supervisors Tommi Lahti

ABSTRACT

The goal of the thesis is to examine the OAuth 2.0 authorization protocol by creating a simple web application with Java and Spring Framework. The idea of the application is to let the user authorize the application to use information stored by Spotify about the user.

Theory part of the thesis includes information about OAuth 2.0 protocol and about different authorization flows that may be used with OAuth 2.0. This thesis is about Authorization code grant flow which is generally recommended for use since it is considered more secure than for example implicit flow. This thesis introduces theory behind Authorization code grant using Spotify API and there is also a small glossary about the terms included in the thesis.

Practice part of the thesis is about introduction to the demo application made for this thesis. The digital libraries and techniques used to create the application are also presented in their own paragraphs. In addition, pieces of actual code are included in relevant chapters.

Keywords OAuth 2.0, authorization, Java

Pages 27 pages including appendices 0 pages

SISÄLLYS

SANASTO

Autentikointi eli *todennus*

on prosessi, jolla varmistetaan se, että käyttäjä todella on kuka väittää olevansa.

Autorisointi eli *valtuutus*

on todennetun käyttäjän valtuuksien tarkastelua. Toisin sanoen voidaan esimerkiksi tarkastella, onko käyttäjällä x valtuuksia muokata tekstiä verkkosivulla.

Käyttäjä

omistaa tietoa, esimerkiksi tämän opinnäytetyön demon tapauksessa Spotifyn palvelimella, ja antaa kolmannelle osapuolelle luvan käyttää omistamaansa tietoa.

REST-rajapinta (tässä työssä käytettävä REST-rajapinta on Spotify API)

on ohjelmointirajapinta, joka antaa mahdollisuuden käyttää ohjelmointirajapintaa tarjoavalla palvelimella olevia resursseja tiettyjen sääntöjen ja käytäntöjen mukaan.

GET ja *POST*

ovat HTTP-protokollan metodeja, joilla sovellukset voivat keskustella keskenään käyttäessään esimerkiksi REST-rajapintaa.

HTTP-protokolla

on yleisesti selainten ja palvelimien käyttämä protokolla. Protokollassa on aina pyyntö ja vastaus. Selaimen web-palvelimelle lähettämä pyyntö sisältää header- ja body-osion. Header sisältää vapaaehtoisia tietoja viestistä kuten käytetyn selaimen, mitä tiedostomuotoja selain voi käsitellä, tietoa halutusta kielestä ja niin edelleen. Yleensä http-pyyntö on GET-muotoinen jolloin body-osiota ei ole mutta esimerkiksi POST-metodilla pyynnön body-osiossa voi olla erilaisia parametreja ja niiden arvoja. HTTP-vastauksessa on niin ikään header ja body. Headerissa voi tulla esimerkiksi tieto onnistuneesta pyynnöstä tai vastaavasti virheilmoitus ja body-osiossa lähetetään varsinainen viesti tai esimerkiksi html-tiedosto.

Base64-koodaus

on tiedon koodausjärjestelmä, jossa tietotavut koodataan tavuiksi jotka jokaisen systeemin pitäisi osata turvallisesti käsitellä.

JSON

on tekstimuotoinen, yksinkertainen tietojen talletus- ja lähetysmuoto. Useissa ohjelmointikielissä on valmiita ohjelmistokirjastoja JSON-muotoisten viestien tekemiseen ja purkamiseen.

URI (*Uniform Resource Identifier*)

on nimi jollekin asialle, esineelle tai vaikka tiedostolle, joka sijaitsee jossain. Se voidaan löytää joko osoitteen tai nimen perusteella. URI kertoo ainoastaan, että tällainen asia jostain löytyy.

URL

on URI, joka myös kertoo mistä tiedosto löytyy verkossa. URL kertoo, miten tiedostoon pääsee käsiksi. Yksinkertaistettuna URL kertoo web-osoitteessa protokollan mitä käytetään (http, ftp, smb…) sekä osoitteen tiedostolle. OAuth -protokollassa yleisesti puhutaan esimerkiksi redirect URI:sta vaikka kyseessä on aina URL. Ollaan kuitenkin turvallisella maaperällä, jos puhutaan URI:sta koska jokainen URL on myös URI. Siispä myös tässä työssä käytetään pääasiassa URI:a.

Endpoint

on URL, johon on rakennettu palvelinsovelluksella jonkinlainen toimivuus. Endpointiin voidaan ottaa yhteyttä ja se palauttaa palvelimen päässä olevia tiedostoja. Esimerkiksi tämän työn demosovelluksen endpoint http://localhost:8080/hitlist -endpoint palauttaa asiakkaan selaimelle tiedoston ”hitlist.html”.

1 JOHDANTO

Idea opinnäytetyön aiheeseen tuli tulevan harjoittelupaikan työtehtäviä sivuten, sekä omasta kiinnostuksesta verkosta löytyvien ohjelmointirajapintojen käyttämiseen. Käyttäjän todentaminen ja tässä tapauksessa tiedostojen oikeuksien antaminen jonkin sovelluksen käyttöön ovat tärkeä osa rajapintojen käyttöä.

Tarkoitus on saada aikaan yksinkertainen sovellus nimeltään Spot-A-Hit, jossa on valmiina tietokannassa Suomen suurimpia hittejä vuosilta 2008—2015. Näitä verrataan Spotify-käyttäjän julkisissa soittolistoissa oleviin kappaleisiin ja saadaan selville montako hittiä miltäkin vuodelta ja sijoitukselta käyttäjän julkisilta soittolistoilta löytyy. Suurta kaupallista menestystä en uskalla sovellukselle luvata eikä sovelluksesta ole tarkoituskaan tehdä julkaisukelpoista esimerkiksi virheentarkastelun ja tietoturvan kannalta. Sovelluksen toiminnallinen puoli on tarkoitus tehdä mahdollisimman paljon itse ohjelmoiden eikä käyttää eri ohjelmointikirjastojen valmiita työkaluja. Tarkoitus on ymmärtää, ja käytännössä esittää, miten kolmannen osapuolen sovellukselle annetaan lupa käyttää sovelluksen käyttäjän Spotify-tiliin liitettyjä tietoja.

Opinnäytetyö tehdään Spotifyn REST-rajapinnan vaatimusten mukaan yrittäen löytää eroavaisuuksia viralliseen OAuth 2.0 -spesifikaatioon mahdollisuuksien mukaan. Suurimpien palveluntarjoajien (mm. Google, Twitter, Facebook, Spotify, Salesforce) tarjoamien rajapintojen käytännöissä on joitain eroavaisuuksia sekä keskenään että virallisen OAuth 2.0 -spesifikaation suhteen, mutta kaikkien eri palveluntarjoajien ohjelmointirajapintojen käytön pitäisi onnistua suhteellisen helposti, kunhan yhden näistä rajapinnoista hallitsee.

2 KÄYTTÄJÄN TUNNISTAUTUMINEN VERKOSSA

Kuten tiskillä, myös verkossa asioidessa on mahdollisuus tunnistaa käyttäjä monella eri tavalla ja vahvuudella. Tunnistuksen vahvuus liikkuu palvelutiskillä henkilökohtaisesti asioidessa passin näyttämisestä esimerkiksi palvelun tietokannassa olevien tietojen, kuten osoitteen tai puhelinnumeron kysymiseen asiakkaalta. Samalla tavalla toimii käyttäjän todentaminen verkossa. Vahvuus vaihtelee esimerkiksi pankkitunnuksilla todentamisesta pelkällä puhelinnumerolla tai käyttäjätunnuksella todentamiseen.

Verkossa asioidessa jokaisella palvelulla on oma käyttäjätunnuksensa ja salasanansa. Tästä seuraa erilaisia käyttäjätunnuksia ja salasanoja kirjoitettuna paperilappuihin ympäri työpöytää tai lompakkoa tai huonoimmassa tapauksessa kaikkiin palveluihin käytetään aina samaa käyttäjätunnusta ja salasanaa. Yhden tietoturvavuodon sijaan onkin vaikkapa kymmenen mahdollisuutta palveluntarjo-

ajan vuotaa käyttäjätunnus ja salasana vääriin käsiin. Tämän ongelman helpottamiseksi on kehitetty palveluja, joissa esimerkiksi Googlen tai Facebookin tunnuksella voidaan kirjautua myös muihin palveluihin. Idea on, että käyttäjä kirjautuu yhteen palveluun ja tämä palvelu vahvistaa toiselle palvelulle, että käyttäjä on kuka väittää olevansa. Käyttäjän käyttäjätunnus ja salasana eivät siis missään vaiheessa päädy toisen palvelun tietoon. Tähänkin voidaan käyttää OAuth 2.0 -protokollaa avuksi ja käyttäjän todentamista yhdellä tunnuksella (autentikointi) käsitellään lisää kappaleessa 4.5.

Tämä työ keskittyy siihen, miten käyttäjä voi antaa sovellukselle luvan käyttää käyttäjän omistamia tietoja, jotka sijaitsevat verkossa toisella palvelimella (autorisointi). Autentikointi ja autorisointi liittyvät kuitenkin paljon toisiinsa, ja käsitteiden ero voi olla monissa tapauksissa aika häilyvä. Jokapäiväisessä verkon käytössä OAuth 2.0 -protokollan käyttö on tavalla tai toisella todella yleistä. On esimerkiksi mahdollista jakaa Ilta-Sanomien artikkeli napin painalluksella vaikkapa Facebookiin. Facebookin käyttämiseen saattaa olla puhelimessa oma sovelluksensa joka myös käyttää tätä juuri samaa tekniikkaa: siinäkin käyttäjä valtuuttaa sovelluksen käyttämään omistamiaan tietoja Facebookin palvelimella.

3 OAUTHIN HISTORIA

OAuth 1.0 kehitys alkoi vuonna 2006 ja se julkaistiin joulukuussa 2007 ja kuten ykkösversioiden kanssa usein käy, myös OAuth 1.0 kärsi aika isoista vioista ja puutteista. Tilannetta paikkaamaan tuli ohjelmistokehys OAuth 2.0 vuonna 2012. Tässä työssä tullaan puhumaan OAuth 2.0 protokollasta joka on nimenomaan autorisointiprotokolla eikä autentikointiprotokolla, jona siihen monessa paikassa saatetaan viitata. OAuth 2.0 todennukseen liittyy monenlaisia autentikointeja käyttäjän ja autorisointiserverin välillä mutta OAuth 2.0 mallintaa ainoastaan sitä, miten käyttäjä voi valtuuttaa kolmannen osapuolen sovelluksen käyttämään käyttäjän palvelimella olevia tietoja välittämättä kuitenkaan käyttäjätunnuksia tai salasanoja. OAuth 2.0 ei esimerkiksi tiedä kuka käyttäjä on, tai mikä hänen sähköpostiosoitteensa tai puhelinnumeronsa on. Raja autorisoinnin ja autentikoinnin välillä on kuitenkin häilyvä, kuten myöhemmin työssäkin tullaan toteamaan. (Richer, 2017)

Ennen OAuth-protokollan kehittämistä sovellusten välisessä kommunikoinnissa asia hoidettiin siten, että sovellus pyysi käyttäjältä toisen sovelluksen käyttäjätunnusta ja salasanaa ja ensimmäinen sovellus saattoi näin kirjautua käyttäjän toiseen sovellukseen (Boyd, 2012). Tämän opinnäytetyön esimerkkejä käyttäen Spot-A-Hit -sovellus olisi pyytänyt käyttäjältä hänen käyttäjätunnustaan ja salasanaansa Spotify-palveluun. Kuten tästä heti voidaan päätellä, toi tämä useita tietoturvallisuusriskejä käyttäjän näkökulmasta katsottuna.

OAuth 2.0 -protokollaa käyttäen tämä toteutetaan siten, että Spot-A-Hit pyytää käyttäjältä lupaa käyttää hänen Spotify-tiliinsä liitettyjä tietoja jotka sijaitsevat

Spotifyn palvelimella. Tämä toteutetaan ohjaamalla käyttäjä Spot-A-Hitin toimesta Spotifyn kirjautumissivulle ja kirjautumalla suoraan Spotify-tiliin. Salasanoja ja käyttäjätunnuksia ei missään vaiheessa luovuteta Spot-A-Hit -sovelluksen käyttöön. (Boyd, 2012)

4 OAUTH 2.0 AUTORISOINTIMALLIT

Oauth 2.0 tuntee ainoastaan kaksi sovellustyyppiä: confidential ja public (Hardt, 2017). Confidential-sovellus pitää käyttäjän tunnukset turvassa esimerkiksi suojatulla web-palvelimella ja public-sovelluksessa taas käsitellään käyttäjän tunnuksia itse sovelluksessa, jolloin se on haavoittuvaisempi esimerkiksi tietomurroille. Vastaavasti OAuth 2.0:ssa yleisimmin käytettyjä ovat seuraavat kaksi autorisointimallia: Authorization code grant (confidential) ja Implicit grant (public). Lisäksi on olemassa kaksi muutakin autorisointimallia, jotka seuraavissa kappaleissa esitellään lyhyesti.

4.1 Authorization code grant

Authorization code grantilla tarkoitetaan yleensä palvelinsovelluksella tapahtuvaa autorisointimallia, joka on yleisesti suositeltu malli paremmasta tietoturvatasostaan johtuen. Tätä mallia nimitetäänkin yleensä ”server side” -malliksi eli palvelinsovellusten käyttämäksi, mutta se ei ole millään muotoa rajattu ainoastaan siihen käyttöön (Boyd, 2012). Vastaavasti Implicit grant -malli ei ole rajattu vain selaimessa pyörivien sovellusten käyttöön. Esimerkiksi tämän opinnäytetyön sovellus olisi hyvin voitu toteuttaa käyttämällä Implicit grant -mallia koska tarkoitus on ainoastaan lukea julkista tietoa kertaluonteisesti. Tekniikoiden samankaltaisuudesta johtuen voidaan sanoa, että jos osaa toteuttaa Authorization code grantin, niin osaa toteuttaa myös Implicit grantin. Tästä syystä opinnäytetyössä valittiin käytettäväksi juuri Authorization code grant.

4.2 Implicit grant

Implicit grant -malli tulee kysymykseen yleensä käytettäessä selainsovelluksia. Näiden sovellusten huono puoli on siinä, että koko tietoliikenne on käyttäjän luettavissa selaimessa. Implicit grantia käytettäessä selainsovelluksessa suositus on, että tietojen käsittelyssä käytettäisiin ainoastaan ”vain luku” -oikeuksia. Tämä lisää tietoturvaa, kun käyttäjän tietoja ei pystytä muokkaamaan. Implicit grantin käyttäjälle näkyvä ero on siinä, että avaimen vanhetessa käyttäjä joutuu uudelleen hyväksymään sovelluksen pääsyn tietoihinsa. Kuitenkin jotkut palveluntarjoajat eivät tätä vaihetta vaadi avainta uudistettaessa, jos käyttäjä on kirjautuneena heidän palveluunsa selaimessa.

4.3 Muut mallit

Lisäksi on olemassa myös Resource Owner password credentials -malli jossa käyttäjä syöttää käyttäjätunnuksen ja salasanan ja sovellus käyttää niitä suoraan tietojen hakemiseen tai muokkaamiseen. Tätä mallia tulisi käyttää tietoturvansa vuoksi ainoastaan palveluntarjoajan virallisissa sovelluksissa. Suurin vaikeus tämän mallin käyttämisessä käyttäjän kannalta on se, miten tunnistaa virallinen sovellus. (Hardt, 2017)

Viimeinen malli on Client Credentials, jossa esimerkiksi sovelluksen käyttämä tietokanta sijaitsee ulkopuolisella palvelimella ja sovellus haluaa pääsyn sinne. Eli tässä tapauksessa sovellus itse omistaa tiedot eikä omistajana ole esimerkiksi ulkopuolinen käyttäjä. (Hardt, 2017)

4.4 Mobiiliapplikaatiot

Mobiiliapplikaatioiden autorisointi putoaa vaihtelevasti näiden väliin. Kaikkiin käyttöjärjestelmiin löytyy jonkinlaiset varastot avaimille, joten jollakin tapaa niitä voitaisiin käsitellä ”confidential” -statuksella. Yleisesti niitä pidetään kuitenkin turvattomina, joten ne menevät ”public” -statukseen. Yleisesti käytetty malli on samanlainen kuin web-sovelluksissakin: jos mobiilisovellus käyttää suojattua palvelinta voidaan käyttää Authorization code grantia, muutoin pitäydytään Implicit grantissa. Lisäksi mobiilisovelluksissa suositellaan kirjautumisessa käytettäväksi natiiviselainta upotetun selaimen sijaan, vaikka se joissain tapauksissa saattaa käyttäjäkokemusta huonontaakin. (Boyd, 2012)

4.5 OpenID Connect

OpenID Connect on OAuth 2.0 -protokollan päällä toimiva autentikointikerros. Toimintaperiaatteena on lisätä olemassa oleviin OAuth 2.0 -parametreihin muutamia uusia parametreja ja näin saada autentikointi onnistumaan. Kuten OAuth 2.0, myös OpenID Connect (virallisemmin versionumeroineen OpenID Connect 1.0) on toinen kehitysversio (aiemmin OpenID 2.0).

Eroavaisuuksia käytettäessä OpenID Connectia pelkän OAuth 2.0 -autorisointimallin lisäksi on aika vähän. OpenID Connctia käytettäessä mukana kulkee sekä access token, että id token. Id token sisältää tietoa käyttäjästä, ja näin saadaan aikaan sovelluksessa käyttäjän tunnistus.

Esimerkiksi pyydettäessä authorization codea lisätään scope-parametriin normaalin Authorization code grantin parametrien lisäksi yksi tai useampia arvoja. OpenID Connect esittelee muitakin mahdollisia lisättäviä parametreja mutta toiminnallisuuden kannalta tärkeimpiä ovat scopen arvot. Scopen mahdollisia arvoja ovat:

Pakollinen arvo *openid*. Sillä kerrotaan palvelimelle, että halutaan tehdä myös OpenID Connect -pyyntö.

Valinnainen arvo *profile*, jolla saadaan käyttäjän profiilista erilaisia tietoja. Nämä tiedot ovat spesifikaation mukaan: name, family_name, given_name, middle_name, nickname, profile, picture, preferred_username, gender, website, birthdate, locale, zoneinfo ja updated_at

Valinnainen arvo *address,* jolla saadaan käyttäjän osoite.

Valinnainen arvo *email*, jolla saadaan käyttäjän sähköpostiosoite. Lisäksi on olemassa *email_verified* -arvo joka on joko tosi tai epätosi riippuen siitä onko palvelu vahvistanut käyttäjän sähköpostiosoitteen oikeaksi. Tähän vahvistusarvoon saamiseen tulee myös oikeus *email*-arvolla.

Valinnainen arvo *phone.* Samoin kuin *email*-arvossa, löytyy myös tästä arvo *phone_verified* ja sen toimintaperiaate on samanlainen.

Valinnainen arvo *offline_access*. Tällä voidaan pyytää access token, joka toimii myös käyttäjän ollessa kirjautuneena ulos sovelluksesta.

Näitä scopen arvoja voi yhdistellä aivan kuten OAuth 2.0:n kyseessä ollessakin. Esimerkiksi näin:

```
?scope=openid profile offline_access
```

Kuva 1. Scope-parametrillä useita arvoja joista ”openid” on ainoa pakollinen.

Tämän jälkeen OpenID Connect -malli etenee samalla tavalla kuin OAuth 2.0 -malli. Erona on ainoastaan se, että sovelluksen käyttöön tulee sekä access token että id token. Tätä id tokenia voidaan käyttää pyydettäessä palveluntarjoajan UserInfo -endpointista annetun scopen mukaisia käyttäjätietoja. (OpenID, 2017)

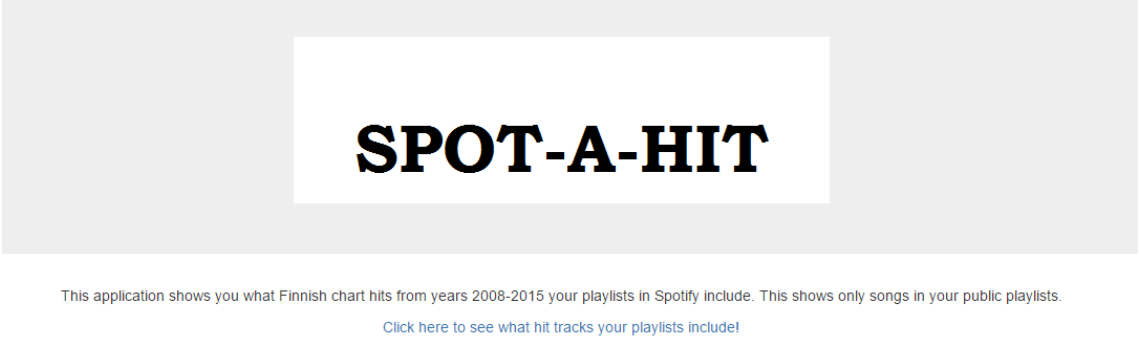
5 DEMOSOVELLUS

5.1 Sovelluksen kuvaus

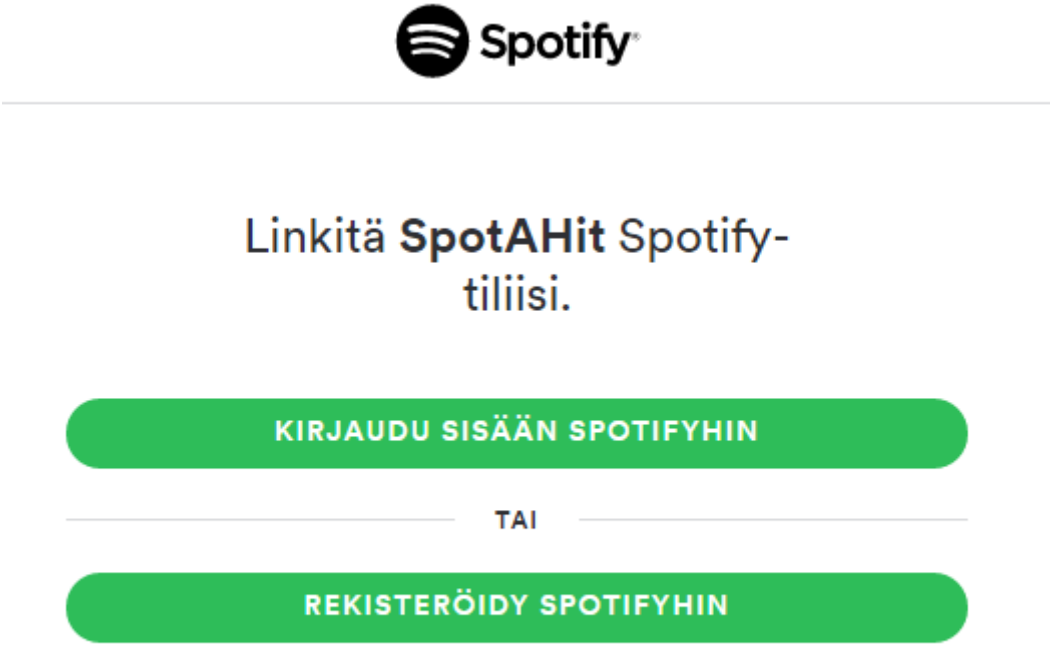
Spot-A-Hit -sovelluksella on tarkoitus hakea Spotifyn rajapinnasta OAuth 2.0 -mallia käyttäen käyttäjän julkiset soittolistat. Tämän jälkeen soittolistoja verrataan sovelluksen tietokannasta löytyviin vuosien 2008-2015 ”suurimpiin hitteihin” Suomessa. Lopuksi käyttäjälle palautetaan tieto siitä, mitkä näistä hiteistä löytyvät käyttäjän soittolistoilta. Suurimmat hitit -tietokanta on kerätty osoitteesta http://suomenvuosilistat.blogspot.fi/2015/09/nain-hittivuosilistat-koottiin.html jossa on myös selostettu laskentatapa suosituimpien kappaleiden saamiseksi vuosittain. Sama toiminnallisuus saataisiin sovellukseen ainoastaan kysymällä käyttäjän Spotify-käyttäjätunnus koska julkiset soittolistat ovat kaikkien nähtävillä ja haettavissa Spotifyn REST-rajapinnasta. Koska kyseessä on kuitenkin

opinnäytetyö, tämä tehtiin Spotify-tilille kirjautumisen kautta. Lisäksi vahvistuksena kirjautumisen onnistumisesta hitlist.html -sivulla näkyy käyttäjän tilin taso (scope: user-read-private). Testaamista varten loin omalle Spotify-tililleni kaksi julkista soittolistaa: ”HAMK” ja ”HAMK kakone”, joista jälkimmäiseen lisäsin neljä sovelluksen tietokannan hittilistalta löytyvää kappaletta.

Valmiin sovelluksen käyttöliittymä selaimessa näyttää tältä:



Kuva 2. Spot-A-Hit -sovelluksen etusivu.



Kuva 3. Etusivun jälkeen aukeaa Spotifyn kirjautuminen, jos käyttäjä ei vielä ole kirjautunut.

Finnish chart hits from 2008-2015 included in nikotas123 playlists:

Account type: premium

Playlist	Artist(s)	Track	Hit in Finland?	Year	Position
HAMK kakone	Kontra	Onomatopoeettinen alkoholiliike	✖	-	-
HAMK kakone	Chisu	Sama nainen	✓	2010	9
HAMK kakone	Train	Drive By	✓	2012	17
HAMK kakone	Edward Maya & Vika Jigulina	Stereo Love	✓	2010	6
HAMK kakone	Happoradio	Hirsipuu	✓	2008	55
HAMK	Satanic Surfers	Soothing	✖	-	-
HAMK	Stam1na	Kadonneet kolme sanaa	✖	-	-
HAMK	Sabaton	1 6 4 8	✖	-	-
HAMK	Eläkeläiset	Hirsipuuhumppa	✖	-	-
HAMK	Within Temptation	Iron	✖	-	-

Kuva 4. Listaus, jossa näkyvät käyttäjän kaikkien julkisten soittolistojen kappaleet sekä se, onko kappale ollut suomessa hittinä millä sijalla ja milloin. Lisäksi näkyy käyttäjän Spotify -käyttäjätunnus ja tilin taso.

5.2 Demosovelluksessa käytetyt tekniikat

Sovellus on ohjelmoitu Javalla käyttäen Spring frameworkin lisäosaa Spring Boot, joka helpottaa ja nopeuttaa web-palvelinsovelluksen kehittämistä huomattavasti. Jo muutamalla rivillä varsinaista koodia saa aikaan ”Hello World!” -sovelluksen REST-toteutuksena – kaikki muu tehdään taustalla Spring Bootin toimesta automaattisesti. Spring framework on enterprise-tasoinen, eli hyvin suuriinkin sovelluksiin skaalautuva Javan framework ja ainakin monien mielestä Javan suosituin framework tällä hetkellä. (Spring Boot, 2017)

Spring Boot tarjoaa suoraan OAuth2-kirjastoa, johon on tehty valmiiksi autorisointia helpottavia komponentteja. Varsinaista testausta en tehnyt, mutta ymmärtääkseni koko autorisointiprosessi, jos se tehdään standardin mukaan, voidaan hoitaa muutamalla rivillä koodia. Tässä työssä kyseinen kirjasto on otettu käyttöön mutta varsinainen autorisointiprosessi on eriytetty omiksi aliohjelmikseen. Näin saadaan parempi tuntuma autorisointiprosessin eri vaiheista sekä siitä, miten mikäkin pyyntö käytännössä rakennetaan.

Sovellus käyttää H2-tietokantaa, joka on Javalla kirjoitettu SQL-perustainen ilmainen tietokanta. H2 voi kirjoittaa tietokannan tiedostoon mutta yleisesti sitä käytetään muistissa toimivana kevyenä ja nopeana testaustietokantana sovellusten kehitysvaiheessa. Muuttaminen mihin tahansa tuotantotietokantaan on helppo prosessi, kun tietokannan palvelu- ja varsinainen toteutuskerros on eriytetty omiksi osikseen. Käytössä Spring Bootin puolelta on Hibernate-kirjasto, joka tarjoaa rajapinnan oliomallin ja tietokannan välillä. Lisäksi CRUD (Create, Read, Update, Delete) -toiminnot luodaan automaattisesti eikä käyttäjän tarvitse tuh-

lata aikaa niiden tekemiseen. Tietokantamalli on valittu sovelluksen jatkokehityksen kannalta. Kyseessä on varsin yksinkertainen sovellus, joten sama toiminnallisuus saataisiin tehtyä helpommin esimerkiksi listoilla.

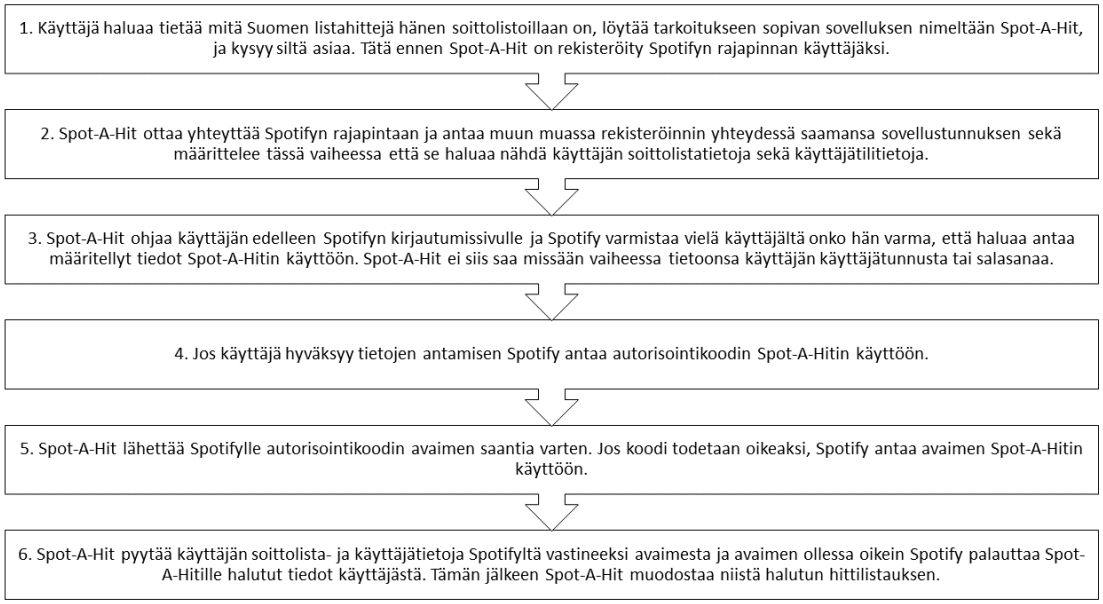
Suurimmat hitit -tietokanta on kopioitu verkosta vuosi kerrallaan Exceliin, josta se sovelluksen käynnistyessä luetaan muistiin H2-tietokantaan. Excel-ratkaisuun päädyin siksi, että Excelissä luetteloa on helppo päivittää lisäämällä vuosia leikkaa-liimaa -tekniikalla sen sijaan että muuntaisi tiedot esimerkiksi csv-muotoon syöttämisen jälkeen. Tietomäärä on kuitenkin niin pieni, että on käytännössä sama, miten tämän osion toteuttaa - puhutaan tiedostojen koosta 1 kilotavun ja 40 kilotavun välillä. Tietojen lukemiseen Spring Bootiin Excelistä on käytetty Apachen POI -kirjastoa, joka on Microsoftin dokumenttien ja Javan väliseen kanssakäymiseen tehty rajapinta. Lisäksi suurimmille hiteille oli tarkoitus hakea Spotifysta suorat Spotify-osoitteet jolloin vertailu olisi ollut helppoa. Tässä kuitenkin tuli mutkia matkaan, koska sama kappale esimerkiksi eri albumilla (vaikkakin saman artistin esittämänä) on Spotifyssa eri kappale. Lisäksi voi olla erilaisia miksauksia kyseisestä kappaleesta, esimerkiksi ”radio edit” joten lopulta päädyin vain vertaamaan artistin nimeä ja kappaletta tietokannan ja Spotifyn antaman tiedon välillä. Todennäköisesti joskus löytyy joku artisti-kappale -yhdistelmä jota ei tunnisteta koska se on väärin kirjoitettu tai väärässä muodossa mutta vielä sellaista ei ole tullut vastaan.

JSON-muotoisten viestien purkaminen on tehty käyttäen Javalle tarkoitukseen tehtyä kirjastoa nimeltään Jackson. Ideana tässä työssä oli se, että teen jokaista tulevaa JSON-muotoista viestiä vastaavan POJOn (Plain Old Java Object) ja otan talteen sieltä ainoastaan niitä tietoja mitä tarvitsen. Esimerkiksi yhden Spotifyn kappaleen tiedoissa on kymmeniä eri arvoja sisäkkäin yksittäisinä arvoina tai taulukkoina ja tarvetta oli ainoastaan nimelle ja esittäjille. Periaate näissä on, että jokaiselle JSONissa olevalle arvolle on luokassa vastine, mutta Jackson osaa jättää huomiotta sellaiset arvot mitä vastaanottavassa luokassa ei ole. Tämä ominaisuus pitää aktivoida erikseen mikä tuntuu vähän ihmeelliseltä koska siinä tapauksessa oletus on, että lähettävän pään JSON ei muutu. Oletuksena on siis, että jos Spotify vaihtaa jonkin parametrin nimeä kappalelistauksessa niin koko Spot-A-Hit -sovellus lakkaa toimimasta. Toinen omasta mielestä kummallinen ominaisuus oli se, että taulukossa ei oletuksena voi olla vain yhtä arvoa vaan tämäkin toiminnallisuus piti Jacksonissa aktivoida erikseen.

Frontendiin, eli käyttäjän näkemään käyttöliittymään, on käytetty Thymeleaf template engineä. Template enginen tehtävänä on yhdistää tietokantamalli sekä web-sivun ulkoasu keskenään. Thymeleafin hyvä puoli on se, että websivun html-koodiin kirjoitetaan sekä ”normaalit” html-attribuutit että Thymeleafin attribuutit ja jos Thymeleaf on toiminnassa niin kyseiset attribuutit jäävät voimaan lopulliselle sivulle. Jos taas Thymeleaf ei ole toiminnassa niin sivu on kuitenkin luettavissa selaimessa rinnalla annetuilla normaaleilla attribuuteilla. Sivun tyyleissä on käytetty Bootstrapia, joka helppo ja nopea tapa tehdä perusvaatimukset täyttäviä ulkoasuja sivustoille.

6 OAUTH 2.0 AUTHORIZATION CODE GRANT DEMOSOVELLUKSESSA

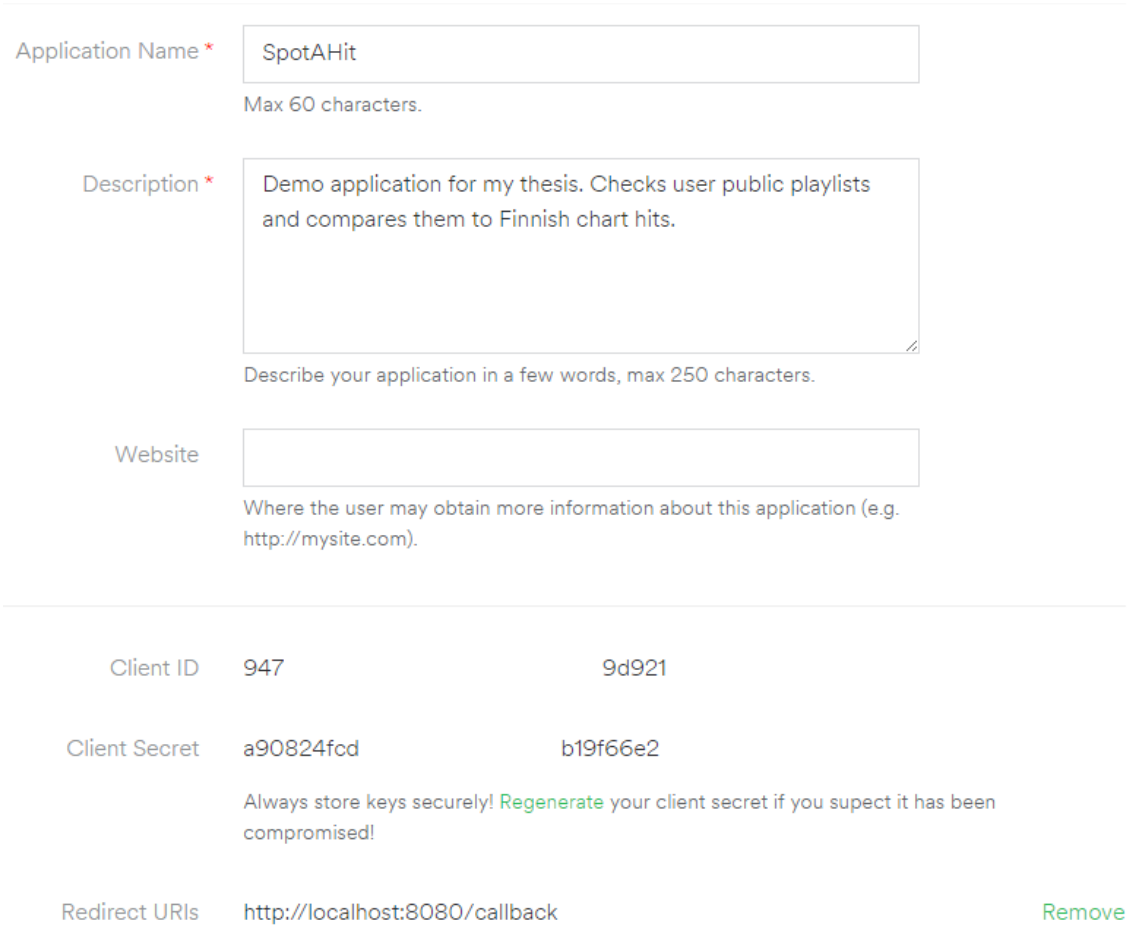
OAuth 2.0 toimintaperiaate on yksinkertaistettuna seuraavassa kuvassa esitettyjen vaiheiden kaltainen. Esimerkissä käytetään tässä työssä demona tehtävää Spot-A-Hit-sovellusta. Kuvassa esiintyviä vaiheita on tarkasteltu lähemmin opinnäytetyön seuraavien alaotsikoiden alla.



Kuva 5. Authorization code grant -todennuksen vaiheet.

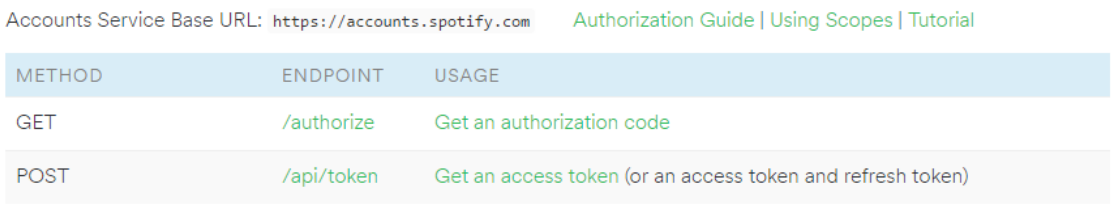
6.1 Spot-A-Hit -sovelluksen rekisteröinti

Ennen varsinaiseen tietojen vaihtoon ryhtymistä Spot-A-Hit-sovellus on rekisteröitävä Spotifyn APIn käyttäjäksi. Rekisteröitäessä sovellusta palveluun tarvitsee yleensä kertoa sovelluksen nimi, pieni kuvaus sovelluksesta sekä uudelleenohjausosoite (*redirect_uri*). Uudelleenohjausosoite on se, mihin palvelun tarjoaja lähettää tietoja kuten authorization coden, access tokenin tai Spotifyn tapauksessa esimerkiksi käyttäjän soittolistat. Monessa paikassa vaaditaan sekä yleiseen hyvään tapaan kuuluu, että uudelleenohjausosoitteet suojataan TLS-protokollalla (HTTPS). OAuth 2.0 -spesifikaatio kuitenkin määrittelee sen ainoastaan sanalla ”pitäisi”, joten sen mukaan pakollista se ei ole eikä tähänkään opinnäytetyöhön liittyvässä demosovelluksessa sitä käytetä. Näiden käyttäjältä vaadittujen tietojen lisäksi palvelun (Spotify) tulee antaa sovellukselle (Spot-A-Hit) yksilöllinen asiakastunnus *ClientID* sekä sovelluksen käytössä oleva varmistuskoodi *Client Secret*.



Kuva 6. Spotifyn sovelluksen rekisteröintisivu. Client ID:stä ja Client Secretistä on peitetty osa pois tietoturvasyistä. Molemmat ovat jatkuvia merkkisarjoja.

Palvelun tarjoaja myös antaa osoitteet omiin endpointeihinsa mihin käyttäjä voi lähettää erilaisia pyyntöjä. Näistä tärkeimmät tunnistautumisen kannalta ovat authorization endpoint ja access token endpoint, joiden käyttöön palataan seuraavissa kappaleissa. Lisäksi Spotify tarjoaa kymmeniä muitakin endpointeja eri toiminnoille kuten kappaleiden etsimiselle tai käyttäjän soittolistan muokkaamiselle. Kaikista työssä käytetyistä parametreista sekä endpointeista löytyy lisää tietoa Spotifyn Web API:sta (https://developer.spotify.com/web-api/)



Kuva 7. Spotifyn authorization- ja token endpointit ja lisäksi tieto siitä, millä http-metodilla kutsut tulee tehdä.

Endpoint	Käyttötarkoitus
http://localhost.com:8080/callback	Redirect URI, johon Spotify ottaa yhteyttä.
http://localhost.com:8080/auth	Koostaa URIn jolla pyydetään Spotifyltä authorization code.
http://localhost.com:8080/hitlist	Hitlist-endpoint, joka palauttaa tiedoston hitlist.html.
http://localhost.com:8080/	Home-endpoint joka palauttaa tiedoston index.html

Kuva 8. Spot-A-Hit -sovelluksen tarjoamat endpointit.

```
clientID = 9479bxxxxxxxxxxxxxxxxxxxxxxxx9d921
clientSecret = a908xxxxxxxxxxxxxxxxcb19f66e2
scopes = user-read-private
responsetype = code
authorizationuri = accounts.spotify.com/authorize
//placeholder
states = 123456qwerty
accesstokenuri = https://accounts.spotify.com/api/token
userinfouri = https://api.spotify.com/v1/me

redirectURI = http://localhost:8080/callback
```

Kuva 9. Spot-A-Hit -sovelluksen käyttämiä muuttujia

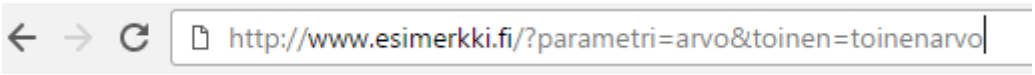
6.2 Spot-A-Hit -sovelluksen pyyntö käyttäjän autorisoinnista

2. Spot-A-Hit ottaa yhteyttää Spotifyn rajapintaan ja antaa muun muassa rekisteröinnin yhteydessä saamansa sovellustunnuksen sekä määrittelee tässä vaiheessa että se haluaa nähdä käyttäjän soittolistatietoja sekä käyttäjätilitietoja.

Spot-A-Hit on palvelinsovellus, joten käyttäjän tunnukset ja tässä tapauksessa lähinnä authorization token pysyy salassa eikä sitä paljasteta käyttäjälle missään vaiheessa. Tämä on tärkeää siksi, että kyseessä on ”confidential” eli luotettu sovellus, joten tässä tapauksessa Authorization code grant -mallia voidaan käyttää. Mahdollista olisi käyttää myös Implicit grant -mallia, koska tiedon tarve on kertaluonteinen. Lisäksi refresh tokenille, eli access tokenin uudistamiselle, ilman käyttäjän varmistusta ei ole tässä sovelluksessa tarvetta. Tässä sovelluksessa käytetään kuitenkin suositeltua Authorization code grant -mallia.

Ensimmäinen pyyntö Spotifylle lähetetään GET-metodilla selaimen osoitekentässä. Spot-A-Hit -sovelluksen etusivulla on linkki, jota klikkaamalla sovellus muodostaa URI:n, jonka parametreja tarkastellaan seuraavassa kappaleessa Spotifyn kannalta.

Parametrien lähettäminen URL:ssa tapahtuu seuraavasti: jos osoitteeseen www.esimerkki.fi haluttaisiin lähettää kaksi parametria, ”parametri” ja ”toinen” ja näiden arvot olisivat vastaavasti ”arvo” ja ”toinenarvo”, se kävisi kirjoittamalla selaimen osoiteriville näin:



Kuva 10. Parametrien lähettäminen osoitteessa. Tunnetaan myös http-header-content-typenä application/x-www-form-urlencoded

Kuva 7 kertoo mihin osoitteeseen sovelluksen tulee lähettää pyynnössä tarvittavat parametrit ja arvot. Spot-A-Hit -sovelluksessa osoitteen luomisen (ja näin ollen myös http://localhost:8080/auth -endpointia) hoitaa itse kirjoitettu AuthController -luokka joka muodostaa osoitteen ennalta määrätyistä parametreista ja hoitaa osoitteeseen siirtymisen redirectillä. Seuraavaksi esitellään kuvat sekä Java-luokan koodista että valmiista URL-osoitteesta.

```
@RequestMapping("/auth")
    public String Auth(UriComponentsBuilder ub) {
        UriComponents uriComponents = UriComponentsBuilder.newInstance()
                .scheme("https").host(authURI)
                .queryParam("response_type", responseType)
                .queryParam("client_id", clientId)
                .queryParam("redirect_uri", redirectURI)
                .queryParam("scope", scopes)
                .queryParam("state", states).build();
        return "redirect:" + uriComponents.toString();
    }
```

Kuva 11. Spot-A-Hit AuthController.java.

```
https://accounts.spotify.com/authorize?response_type=code&client_id=9479ba                b9d921
&redirect_uri=http://localhost:8080/callback&scope=user-read-private&state=123456qwerty
```

Kuva 12. Ensimmäinen kutsu Spotifyn authorization endpointiin. Client ID:stä on peitetty osa.

Mahdollisia osoitteen mukana lähetettäviä parametreja Spotifyn /authorize endpointiin tässä vaiheessa ovat (https://developer.spotify.com/web-api/authorization-guide/):

Pakollinen parametri *response_type*. Tässä tapauksessa ainoa arvo tälle parametrille on ”code”. Se tarkoittaa sitä, että mallina on Authorization code grant.

Pakollinen parametri *client_id*. Tämä on uniikki tunnus, minkä Spotify Spot-A-Hitille loi. Tällä tunnistetaan ohjelma palvelun tarjoajan päässä.

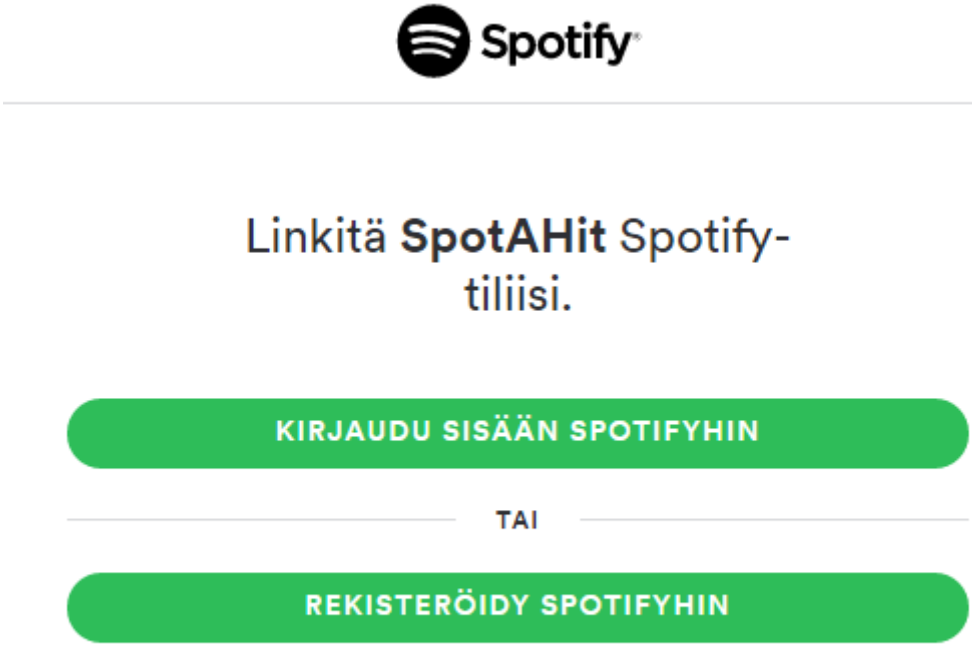
Pakollinen parametri *redirect_uri*. Tämä on sama, mikä on ilmoitettu palvelun tarjoajalle rekisteröitäessä sovellusta ja pitää olla täsmälleen samassa muodossa. Yleensä tämä on vapaasti muutettavissa ja ainakin Spotify tarjoaa mahdollisuuden monelle vaihtoehdolle samanaikaisesti. HUOM! Redirect_uri ei ole OAuth 2.0 -spesifikaatiossa pakollinen parametri, vaikka sitä suurin osa palvelun tarjoajista, kuten Spotify, pakollisena käyttääkin.

Valinnainen parametri *scope*. Tämä määrittelee mitä tietoja käyttäjä haluaa sovellukselle luovuttaa. Esimerkkinä tässä sovelluksessa on annettu scopelle arvo ”user-read-private”, joka antaa luvan tarkastella käyttäjän Spotify-tilauksen tilaa (free, premium…). Erilaisia scopeja Spotifyltä löytyy noin kymmenen ja huomattavaa on, että esimerkiksi käytettäessä Implicit grantia on tietoturvan vuoksi suositeltavaa käyttää ainoastaan read-scopeja jolloin käyttäjän tietoja ei päästä muokkaamaan.

Valinnainen parametri *state*. Tämä on tehty estämään lähinnä Cross Site Request Forgeryä (CSRF). State-parametrillä voidaan kuitenkin ehkäistä tätä luomalla käyttäjälle esimerkiksi joka sessioon uniikki state-parametri. Lisää tietoa CSRF:stä löytyy esimerkiksi OWASP-järjestön sivuilta https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

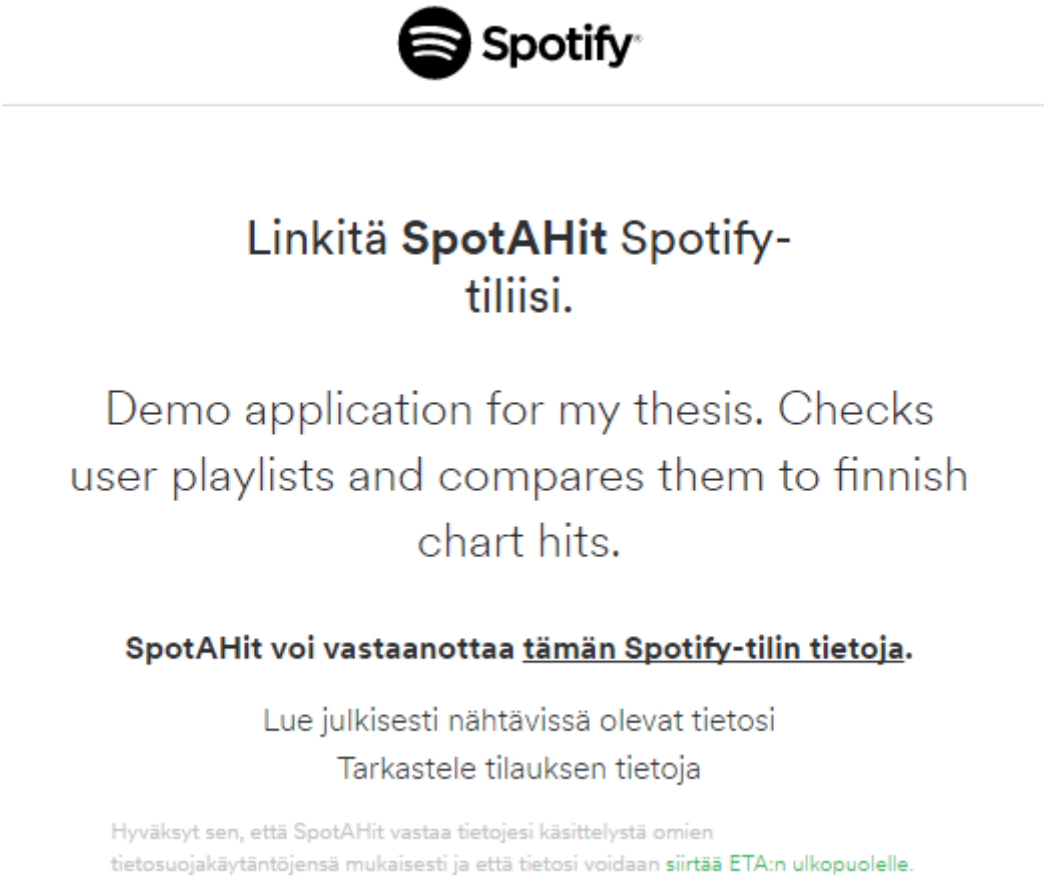
6.3 **Palveluun kirjautuminen**

Tässä vaiheessa edellisen vaiheen URIn lähettämisen jälkeen avautuu selaimessa Spotifyn kirjautumissivu. Huomattavaa on, että URI ja sen parametrit ovat tässä vaiheessa käyttäjän luettavissa selaimen osoiterivillä. Mitään salaista tietoa ei siis lähetetä, eikä näitä arvoja muokkaamalla aiheuteta mitään vahinkoa. Kuitenkin OAuth 2.0 -spesifikaation mukaan parametrin *callback_uri* arvoa ei ole pakko rekisteröidä ja näin ollen niitä ei ole pakollista myöskään verrata toisiinsa. Yksi väärinkäytöksen mahdollisuus tässä kohdassa onkin se, että jos sitä ei ole sovelluksen rekisteröinnin yhteydessä annettu, voisi käyttäjän authorization code mennä tässä vaiheessa vääriin käsiin muuttamalla osoitteessa olevaa parametrin arvoa. Authorization codella ei sinällään tee vielä mitään, koska access tokenin saamiseksi tarvitaan myös client secret, mutta sama periaate pätee jo autorisoidun käyttäjän kohdalla access tokenin saamiseksi.



Kuva 13. Spotifyn kirjautumissivu aukeaa klikkaamalla Spot-A-Hitin etusivun linkkiä joka johtaa Spot-A-Hitin /auth endpointiin.

Vaihtoehtoisesti käyttäjä voi olla jo kirjautunut Spotifyhin, jolloin selaimeen aukeaa suoraan kysely, jossa pyydetään käyttäjältä Spot-A-Hit sovelluksen linkittämistä Spotify tiliin. Sovelluksen linkittäminen tapahtuu ainoastaan kerran, ja linkityksen voi poistaa ainoastaan Spotify-tilin asetuksista. Lisäksi on huomattavaa, että jos tässä vaiheessa annetaan sovellukselle salliva scope, esimerkiksi luku- ja kirjoitusoikeus soittolistaan, niin se säilyy niin kauan, kunnes käyttäjä poistaa sovelluksen hyväksynnän Spotifyn tiliasetuksista. Vaikka sovellus jossain vaiheessa rajottaisi omaa scopeaan poistamalla itseltään kirjoitusoikeuden niin kirjoitusoikeus kuitenkin sovelluksella säilyy. Jos taas oikeuksia halutaan lisää, tulee siitä käyttäjälle huomautus muuttuneine oikeuksineen. En tiedä miten yleinen käytäntö tällainen on, mutta omasta mielestä scopen pitäisi olla dynaaminen molempiin suuntiin aina kun authorization codea pyydetään. Ehkä taustalla on joku palveluaspekti, jossa halutaan vähentää käyttäjältä vaadittavia klikkauksia ja uudelleenohjauksia.



Kuva 14. Spot-A-Hitin linkittäminen Spotify-tiliin. Tässä näkyy myös haluttu scope, eli julkiset tiedot joka on oletuksena aina sallittu vaikka mitään scopea ei olisi määritelty ja tilauksen tiedot eli scope-parametrin arvo user-read-private (Kuva 12).

Jos, ja kun käyttäjä tämän vaiheen hyväksyy palauttaa Spotify Spot-A-Hitille authorization coden. Authorization code on nimensä mukaisesti se, mikä erottaa Authorization code grantin ja Implicit grantin toisistaan. Implicit grant on muuten lähes identtinen, mutta seuraava authorization code -vaihe jää pois eli Implicit grantissa Spotify antaisi suoraan tässä vaiheessa access tokenin sovelluksen käyttöön.

6.4 Authorization code

Tässä vaiheessa huomion arvoista on se, että Spotify palauttaa authorization coden suoraan palvelinsovellukselle. Käyttäjä ei missään välissä näe tätä koodia tai pääse käsiksi tähän koodiin. Authorization code on kertakäyttöinen. Jos sitä yritetään käyttää toista kertaa, tulisi palvelun palauttaa virhe sekä mahdollisesti peruuttaa jo sovelluksen saamat access tokenit. Lisäksi OAuth 2.0 suositus on, että authorization coden elinaika on maksimissaan 10 minuuttia ja koodi onkin

tarkoitus tietoturvasyistä käyttää heti sen saamisen jälkeen. Spot-A-Hit -sovelluksen tapauksessa palvelinsovelluksen tulee käyttää authorization code access tokenin pyytämiseen 10 minuutin sisällä. 10 minuuttia on nykymaailmassa todella pitkä aika käyttäjän odottaa sovelluksen toiminnallisuutta, joten voimassaoloaika voisi olla oletuksena paljon lyhyempikin.

Varsinaisen authorization coden pyynnön tekeminen esiteltiin kappaleessa 6.2 ja vastaus pyyntöön tulee yksinkertaisesti URI-osoitteena, jossa palautuu myös parametreja ja niiden arvoja. Vastaus palautuu *redirect_uri* -parametrissa määriteltyyn endpointiin, joka Spot-A-Hitin tapauksessa on http://localhost:8080/callback (Kuva 8). Vastauskoodiin on kaksi vaihtoehtoa: joko käyttäjä antaa sovellukselle luvan käyttää tietoja, tai sitten lupaa ei anneta. Käyttäjän antaessa sovellukselle luvan käyttää tietojaan palautuu osoitteessa kaksi parametria:

Pakollinen parametri *code*. Tämä on authorization code, joka on vaihdettavissa tietyin ehdoin access tokeniksi.

Valinnainen parametri *state*. Tämä on huijauksenestoparametri, joka pitää tarkistaa, ja jonka tulee olla sama kuin autorisointipyyntöä lähetettäessä.

```
http://localhost:8080/callback?code=AQDrxFSUcsHOMxGoXIwtWat0gsjWyH2rBxk1-Y6pppIJzUkDGFgpUmS5cmD9A-
nHs_BGaMOHu5Y_m2CZdI9qY7n7Cxv4ASEKIxarmGA1Q4AjimupaBRW5ZydwdnZHR2s9bseAUiCcwl-SXEotVxaOjdLpdFMPZmlCOzXiNAQYzHZvfBmCdrQJOV-
bhj6sgji4Hz8isrxsnlXcazMJXc0byyZ_ZkLUKs&state=123456qwerty
```

Kuva 15. Esimerkki Spotifyn muodostamasta osoitteesta redirect URI:n määrittelemään endpointiin. Sekä code- että state-parametrit löytyvät. Tämä osoite on Spotifyn vastaus Spot-A-Hitin lähettämään authorization code -pyyntöön.

Spot-A-Hit -sovelluksen toiminta on aina samanlainen, joten URI-osoitteen käsittelyn callback endpointissa tekevä controller hoitaa myös muut vaadittavat toiminnot ennen hittilistan tulostamista käyttäjälle. Seuraavassa on esitelty kyseisen itse kirjoitetun CallbackController.java -luokan koodi joka poimii Spotifyn lähettämästä URI:sta parametrit, tarkistaa state-parametrin samaksi kuin lähetettäessä ja seuraavaksi kutsuu itse kirjoitetut metodit, jotka järjestyksessä hakevat access tokenin, käyttäjän tiedot, soittolistat sekä yksittäiset kappaleet.

```
@RequestMapping(value = "/callback", method = RequestMethod.GET)
    //medodille pitää tulla parametrit code ja state
    public String callback(@RequestParam(value="code", required = true) String authCode,
                           @RequestParam(value="state", required = true) String state,
                            RedirectAttributes redirectAttr) throws IOException {
        //tarkistetaan että state on sama kuin lähetettäessä
        //tässä sovelluksessa state on aina sama, joten ongelmaa ei tule
        if (stateOriginal.equals(state)) {
            SpotifyAccessToken sat = fetchToken(authCode, state);
            UserInfo userInfo = fetchUserInfo(sat);
            Playlists pl = fetchPlaylists(userInfo, sat);
            fetchUserTracks(pl, sat, userInfo);
        }
        else {
            //tänne tulisi toimet, jos state on eri kuin lähetettäessä
        }

        redirectAttr.addFlashAttribute("user", userMap);
        return "redirect:hitlist/";
    }
```

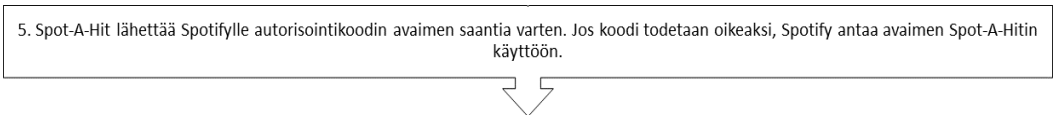
Kuva 16. Spot-A-Hit -sovelluksen tapa käsitellä Spotifyltä saapuva edellisen kuvan (Kuva 15) URI-osoite. CallbackController.java.

Toinen vaihtoehto luvan antamiselle ja onnistumiselle on se, että tapahtuu jokin virhe tai käyttäjä ei anna sovellukselle lupaa käyttää tietojaan. Siinä tapauksessa palautuu myös kaksi parametria:

Pakollinen parametri *error*, jossa kerrotaan mikä meni vikaan. Parametrin arvona voi olla esimerkiksi ”access_denied” tai ”invalid_scope”.

Valinnainen parametri *state* on toiminnaltaan sama kuin onnistuneessa yrityksessä.

6.5 Access token



Tämä vaihe on teoriassa varsin suoraviivainen. OAuth 2.0:n (kuten myös Spotifyn) mukaan access tokenin POST-pyyntö tehdään Spotifyn token endpointiin. Kuulostaa yksinkertaiselta mutta mitä tämä käytännössä tarkoittaa?

Aloitetaan helpoimmasta eli Spotifyn token endpointista. Tämän endpointin URL, eli osoite mihin pyyntö tehdään, sekä käytettävä metodi on esitelty Kuva 7.

Seuraavaksi POST-pyyntö: http-protokollassa on viestissä aina olemassa pyyntö ja vastaus. Tässä tapauksessa Spot-A-Hit lähettää pyynnön POST-metodilla Spotifylle ja siihen saapuu vastaus. POST-tyyppisessä pyynnissä parametrit lähetetään body-osiossa, kun taas GET-pyynnissä ne lähetetään osoitteessa (Kuva 12). Lähettääkseen access token -pyynnön tarvitsee yksinkertaistettuna tietää vain,

että http-viestissä on otsikko (header) ja runko (body), ja että niihin voidaan asettaa erilaisia arvoja ja parametreja.

Spotifyn access token -pyynnössä käytetyt parametrit ovat seuraavat:

Pakollinen parametri *Authorization*, joka voidaan lähettää kahdella eri tavalla. Suositeltava tapa on lähettää se headerissa authorization-kentän arvona muodossa ”Authorization: <type> <credentials>. Type on yleensä Basic ja niin myös Spotifyn tapauksessa. Credentials-arvoksi tulee sekä client_id että client_secret, ja Spotify vaatii ne erotettuna kaksoispisteellä ja base64-koodattuna. Lopullinen muoto on siis tässä tapauksessa: ”Authorization: Basic <base64 client_id:client_secret>. Jos parametrin arvon lähettäminen headerissa on syystä tai toisesta mahdotonta toteuttaa on client_id ja client_secret myös mahdollista lähettää muiden parametrien mukana body-osassa koodaamattomina. Tämä tapa on joissakin OAuth 2.0 -protokollaa käyttävissä palveluissa jopa ainoa mahdollinen.

Pakollinen parametri *redirect_uri*. Tätä käytetään ainoastaan pyynnön varmistamiseen, ei uudelleenohjaukseen. Tämän pitää olla sama kuin authorization codea pyydettäessä ja parametri lähetetään viestin body-osassa.

Pakollinen parametri *code*. Tämä on authorization code, joka saatiin edellisessä vaiheessa. Myös tämä parametri lähetetään viestin body-osassa.

Pakollinen parametri *grant_type*. Tässä tapauksessa siinä tulee olla arvo ”authorization_code” joka kertoo, että tässä tapauksessa käytetään autorisoinnissa Authorization code grantia. Edellisten tapaan tämä parametri lähetetään viestin body-osassa. Spotify API: tukemat kolme autorisointitapaa ovat: Authorization code grant, Client credentials ja Implicit grant. (https://developer.spotify.com/web-api/authorization-guide/)

POST-pyynnön rakentaminen onnistuu Javalla esimerkiksi seuraavalla tavalla: ensin rakennetaan header, seuraavaksi body ja lopuksi ne yhdistetään yhdeksi kokonaisuudeksi, joka sitten lähetetään Spotifylle token endpointiin. Seuraavassa kuvassa Spot-A-Hit -sovellukseen tehty koodi:

```
//headerissa sisällön tyyppi application/x-www-form-urlencoded
//tarkoittaa siis, että parametrit on bodyssä esitelty kuten URIssa
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);

//headerissa pitää olla client_id ja client_secret base64 encoded
//Authorization: Basic <base64 encoded client_id:client_secret>
String idSecret = clientId + ":" + clientSecret;
String encoded = "Basic " + Base64.getEncoder().encodeToString(idSecret.getBytes( charsetName: "utf-8"));
headers.set("Authorization", encoded);

//bodyyn uri-muodossa parametrit
String body = "redirect_uri=" + redirectURI + "&code=" + authCode + "&grant_type=authorization_code";

//luodaan http-entity headerista ja bodysta
HttpEntity<String> entity = new HttpEntity<>(body, headers);
```

Kuva 17. Access tokenin POST-pyynnön rakentaminen.

Ja kuten http-protokollaan kuuluu, pyyntöön tulee myös vastaus. Jos kaikki menee hyvin, vastauksen statuskoodi header-osassa on arvo 200 eli kaikki kunnossa. Varsinainen access token ja muut parametrit tulevat vastauksena body-osiossa json-muotoisena. Palautuvat parametrit ovat seuraavat:

Pakollinen parametri *access_token* on avain, jolla sovellus voi pyytää palvelimelta käyttäjän myöntämän scopen mukaan sovelluksen käyttöön antamia tietoja.

Pakollinen parametri *refresh_token* on keino hakea uusi access token palvelimelta, kun edellinen access token vanhenee. Spotifyn tapauksessa palvelimelle lähetetään samanlainen POST-pyyntö kuin muutoinkin access tokenia pyydettäessä, mutta authorization coden tilalle laitetaan refresh token. Eli toisin sanoen käyttäjän, eikä sovelluksen, ei tarvitse pyytää uudelleen authorization codea. Joillakin palveluntarjoajilla on myös tapana käyttäjän vaihtaessa salasanaa, perua kaikki refresh tokenit, mikä lisää osaltaan tietoturvaa.

Pakollinen parametri *token_type* on Spotifyllä aina muotoa ”bearer”. Näin myös oletuksena suurimmassa osassa muita palveluja. Vapaa suomennos tästä tyypistä menee jotenkin ”anna tämän tokenin haltijalle (bearer) oikeudet”. Muunkin tyyppisiä tokeneja on olemassa, kuten esimerkiksi MAC. (https://tools.ietf.org/html/rfc6749)

Pakollinen parametri *scope* on sama kuin aiemminkin, ja arvot tulevat listan muodossa muistutuksena tälle access tokenille annetuista oikeuksista. Tämä ei ole OAuth 2.0 -spesifikaation mukaan pakollinen parametri, jos mikään ei ole muuttunut, mutta scopen muuttuessa tämä on pakollinen parametri.

Pakollinen parametri *expires_in* on kokonaislukutyyppinen parametri, joka kertoo ajankohdan, jolloin lähetetty access token vanhenee. Arvo on annettu sekunneissa ja oletusarvona Spotifyllä se on 3600, eli yksi tunti. Refresh token ei kuitenkaan vanhene ennen kuin käyttäjä poistaa linkityksen Spotify-tilin ja Spot-A-Hit -sovelluksen välillä, eli se on voimassa periaatteessa ikuisesti.

Kaikki ei kuitenkaan mene aina suunnitellusti ja myös tässä vaiheessa on mahdollista saada virheilmoitus. Tässä vaiheessa se tulee header-osiossa http-statuskoodilla 400 (bad request). Virheviestissä on mukana seuraavanlaisia parametreja:

Pakollinen parametri *error*, jonka arvona voi olla *invalid_client*, *unauthorized_client*, *invalid_request*, *invalid_grant*, *unauthorized_grant_type* tai *invalid_scope*. Arvot ovat aika yksiselitteisiä, mutta lisää tietoa löytyy esimerkiksi OAuth 2.0 -spesifikaatiosta.

Valinnainen parametri *error_uri*, jonka sisältämästä osoitteesta saa lisätietoja virheestä. Selityksen pitäisi olla kansantajuinen, kuten myös seuraavan parametrin.

Valinnainen parametri *error_description*, joka sisältää ei-tietokonekielisen kuvauksen tapahtuneesta virheestä.

6.6 Pyydettyjen tietojen palautus ja käsittely sovelluksessa

6. Spot-A-Hit pyytää käyttäjän soittolista- ja käyttäjätietoja Spotifyltä vastineeksi avaimesta ja avaimen ollessa oikein Spotify palauttaa Spot-A-Hitille halutut tiedot käyttäjästä. Tämän jälkeen Spot-A-Hit muodostaa niistä halutun hittilistauksen.

Edellisessä vaiheessa vastauksena tulleella access tokenilla on aluksi tarkoitus hakea käyttäjän tilitiedot scope-parametrin arvon *user-read-private* mukaisesti. Käyttäjän tiedoista saadun Spotify-käyttäjätunnuksen perusteella haetaan seuraavassa vaiheessa käyttäjän julkiset soittolistat, ja tämän jälkeen viimeisellä haulla haetaan erikseen jokaisen soittolistan kappaleet.

Käyttäjätiedot haetaan Spotifyn ohjeiden mukaan lähettämällä GET-pyyntö osoitteeseen https://api.spotify.com/v1/me. Pyynnön header-osassa lähetetään Authorization -kentässä arvona Bearer ja access token. Muoto on siis Authorization: Bearer <access token>. Tämän jälkeen Spotify palauttaa access tokenin scopen mukaiset tiedot JSON-muodossa ja näistä sovellus lukee halutut tiedot malliobjektiin. GET-pyyntöä ei tässä tapauksessa lähetetä selaimen osoiterivillä kuten aiemmin, vaan käytetään Spring frameworkin tarjoamaa httpentity-luokkaa. Spot-A-Hit:ssä otetaan Spotifyn lähttämästä viestistä UserInfo-luokkaan ylös käyttäjän maa, profiilin osoite, tilin taso sekä käyttäjätunnus. Tässä sovellusversiossa tietoa käyttäjän maasta ei käytetä.

Jos haluttaisiin tietoon myös käyttäjän sähköpostiosoite, pitäisi scopessa olla arvona myös *user-read-email,* mutta Spotify ei millään tapaa varmista tai takaa käyttäjän sähköpostiosoitteen oikeellisuutta.

```
public UserInfo fetchUserInfo(SpotifyAccessToken sat) throws IOException {
        //lähetetään uusi GET-pyyntö käyttäjän id:n saamiseksi
        HttpHeaders headers = new HttpHeaders();
        headers.set("Authorization", "Bearer " + sat.getAccess_token());

        HttpEntity<String> entity = new HttpEntity<>("", headers);
        ResponseEntity<String> response = restTemplate.exchange(userInfoURI, HttpMethod.GET, entity, String.class);
        String token = response.getBody();

        UserInfo userInfo = new UserInfo();
        ObjectMapper mapper = new ObjectMapper();
        mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, Boolean.FALSE);
        userInfo = mapper.readValue(token, UserInfo.class);

        //tallennetaan userMappiin käyttäjän tunnus ja tilin taso
        //hitlist-sivua varten
        userMap.put("user", userInfo.getId());
        userMap.put("product", userInfo.getProduct());
        return userInfo;
    }
```

Kuva 18. Spot-A-Hit-sovelluksen metodi käyttäjän tietojen hakemiseen.

Tämän jälkeen haetaan käyttäjän julkiset soittolistat. Pyyntö tehdään Spotifyn ohjeistuksen mukaan osoitteeseen https://api.spotify.com/v1/me/playlists. Header-osioon laitetaan samalla tavoin kuin edellisessä vaiheessa Authorization

-kentälle arvoksi Bearer <access token>. Lisäksi parametrilla *limit* on soittolistojen määrää mahdollista rajoittaa välillä 1—50. Oletusarvona soittolistojen määrä on 20. Jos soittolistoja on enemmän kuin 50, on toiseksi parametriksi pyyntöön mahdollista laittaa *offset,* joka kertoo ensimmäisen palautettavan soittolistan indeksin. Jos soittolistoja on 333, on ne kaikki käytävä maksimissaan 50 kerrallaan läpi, kunnes soittolistoja palautuu nolla kappaletta. Mistään en löytänyt mainintaa soittolistojen järjestyksestä tai siitä, palautuvatko ne aina samassa järjestyksessä kyselyn tekijälle, jos soittolistoja ei muuteta. Tässä palautuvan JSON:n rakenteessa alkaa olemaan jo sisäkkäisiä listoja, joten malliobjektin kanssa tulee olla tarkkana. Spot-A-Hit -sovelluksessa talteen otetaan Playlist.java -luokkaan soittolistan nimi sekä soittolistan osoite Spotifyn rajapinnassa.

```
private Playlists fetchPlaylists(UserInfo userInfo, SpotifyAccessToken sat) throws IOException {
    String playlistURI = userInfo.getHref() + "/playlists";
    HttpHeaders headers = new HttpHeaders();
    headers.set("Authorization", "Bearer " + sat.getAccess_token());

    HttpEntity<String> entity = new HttpEntity<>("", headers);
    ResponseEntity<String> response = restTemplate.exchange(playlistURI, HttpMethod.GET, entity, String.class);
    String token = response.getBody();

    ObjectMapper mapper = new ObjectMapper();
    mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, Boolean.FALSE);
    mapper.configure(DeserializationFeature.ACCEPT_SINGLE_VALUE_AS_ARRAY, Boolean.TRUE);
    Playlists playLists = mapper.readValue(token, Playlists.class);

    return playLists;
}
```

Kuva 19. Spot-A-Hit -sovelluksen metodi käyttäjän soittolistojen hakemiseen.

Viimeinen tehtävä on hakea kappaleiden nimet ja esittäjät soittolistojen mukaan ja tallettaa nämä tietokantaan. Lisäksi tietokantaan tallennetaan jokaisen kappaleen kohdalle käyttäjän käyttäjätunnus, jotta soittolistat ja kappaleet voidaan yksilöidä. Periaate on täysin samanlainen kuin edellisissäkin vaiheissa, http-pyyntöviestin header-osassa lähetetään access token, ja pyyntö tehdään edellisessä vaiheessa talletettuun osoitteeseen https://api.spotify.com/v1/users/{user_id}/playlists/{playlist_id}/tracks. Tämän osoitteen Spotify tarjoaa parametrina soittolistan ottamisen yhteydessä ilman /tracks -päätettä, joten sen muokkaaminen on helppoa. JSON-muotoinen vastaus yksittäisestä kappaleesta on varsin sekava ja monitasoinen, mutta rajapinta tarjoaa *fields* -parametrilla mahdollisuuden suodattaa vain halutut parametrit vastaukseen mukaan. Lisäksi kuten edellisessäkin vaiheessa, niin tässä vaiheessa on myös mahdollisuus *limit*- ja *offset*-parametrien käyttöön.

```
private void fetchUserTracks(Playlists pl, SpotifyAccessToken sat, UserInfo userInfo) throws IOException {
    //ensin tyhjennetään kannasta mahdollinen vanha soittolista
    userSongService.deleteAll();

    HttpHeaders headers = new HttpHeaders();
    headers.set("Authorization", "Bearer " + sat.getAccess_token());

    HttpEntity<String> entity = new HttpEntity<>("", headers);

    List<Playlists.Items> lists = pl.items;
    ObjectMapper mapper = new ObjectMapper();
    mapper.configure(DeserializationFeature.ACCEPT_SINGLE_VALUE_AS_ARRAY, Boolean.TRUE);
    mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, Boolean.FALSE);

    for (Playlists.Items item:lists) {
        ResponseEntity<String> response = restTemplate.exchange(item.getHref() + "/tracks", HttpMethod.GET, entity, String.class);
        String token = response.getBody();
        TrackInfo trackInfo = mapper.readValue(token, TrackInfo.class);
        UserSong userSong = new UserSong();

        //Artisteista lista
        List<String> artistNames = new ArrayList<>();
        for (TrackInfo.Items items:trackInfo.items) {
            //kappaleen nimi kantaan
            userSong.setSong(items.track.name);
            //playlistan nimi kantaan
            userSong.setPlaylist(item.getName());
            //lisätään kaikki artistit listaan
            for (TrackInfo.Items.Track.Artist name:items.track.artists) {
                artistNames.add(name.name);
            }

            //lopuksi artistit stringiksi ja kantaan
            StringBuilder builder = new StringBuilder();
            for (String name:artistNames) {
                if (builder.length() != 0) {
                    builder.append(" & ");
                }
                builder.append(name);
            }
            userSong.setArtist(builder.toString());
            userSong.setPlaylistOwner(userInfo.getId());
            userSongService.saveOrUpdate(userSong);
            userSong = new UserSong();
            artistNames = new ArrayList<>();
        }
    }
}
```

Kuva 20. Spot-A-Hit -sovelluksen metodi jossa artistit ja kappaleet lisätään tietokantaan vertailun helpottamiseksi.

Lopuksi hittien ja käyttäjän kappaleiden vertaaminen tehdään controllerissa, joka hoitaa Spot-A-Hit -sovelluksen /hitlist-endpointia. Vertaamiseen, ja näin ollen myös kappaleen sivulla näyttämiseen, on tehty oma luokkansa joka sisältää soittolistan, esittäjän, kappaleen nimen, onko kappale ollut hitti suomessa, sekä mahdollisen vuoden ja sijoituksen.

7 OPINNÄYTETYÖN OPETUKSET JA POHDINTAA

Ensimmäinen ajatus, kun uutta asiaa alkaa opiskella on se, että miksi tieto on niin hajallaan pitkin ja poikin tiedon valtaväylillä. Onko oma ajattelumalli tai tiedon tarve niin erilaisia kuin muilla, ettei minkäänlaista rautalankamallia ole olemassa? Toisaalta kysymykset ovat yleensä aika sovelluskohtaisia kuten tässäkin tapauksessa ja näinkin kapea-alaisen asian esittämiseen tarvitsi etsiä tietoa ainakin kymmenen muunkin tekniikan saralta, että maaliin asti pääsi. Tiedon löytämistä luulisi helpottavan se, että OAuth 2.0 -protokollaa kuitenkin käytetään enenevissä määrin ja samoin Spring on ehkä Javan eniten käytetty framework web-sovelluksien tekoon. Eihän se toisaalta pakota käyttäjiä kirjoittamaan omia kokemuksiaan ja tietojaan internetiin asti. Lisäksi esimerkiksi Google tarjoaa oman Java-kirjastonsa OAuth 2.0:n käyttöön joten ehkä olisi ollut helpompiakin

tapoja päästä tänne pohdintaosioon asti. Seuraavissa kappaleissa muutamia omasta mielestä tärkeimpiä nostoja eteen tulleista aiheista:

OAuth 2.0 on käsitteenä ja teoriatasolla helposti ymmärrettävä kokonaisuus. Käytännön toteutus osoittautui hieman mutkikkaammaksi esimerkkien puutteen vuoksi. Itselleni lähes kaikki ohjelmointimaailmassa on tässä vaiheessa uutta, joten ehkä vaikkapa joku pieni http-protokollan läpikäynti alkuun olisi hyvä ja siihen yritin myös tässä opinnäytetyössä huomiota kiinnittää. Monta kertaa tähän työhön liittyvää demosovellusta tehdessä on tullut jo ”ai se tarkoittaa vain tätä” tai ”ai se oli näin helppo” -tyyppisiä tuntemuksia. Toki tämä johtuu paljolti siitä, että käytännössä minkäänlaista ohjelmointirutiinia ei ole, tai ei vastaavasti tiedä miten jossain muussa ympäristössä asia toteutetaan ja siksi ei osaa lähteä hakemaan samankaltaista ratkaisumallia ongelmiin.

Autorisoinnin ja autentikoinnin eroja en ollut ikinä oikein ennen tätä opinnäytetyötä edes ajatellut. Vieläkin on vähän vaikeaa ymmärtää miksi tämä on pelkästään autorisointia koska tietyn ihmisen tietojahan tässä aina käsitellään. Missä vaiheessa autorisointitarve muuttuu autentikointitarpeeksi? Jos esimerkiksi Facebook-sovelluksen puhelimeen voi tehdä vain autorisoimalla niin voiko esimerkiksi sähköpostisovelluksen tehdä samalla tavalla. Entä pankkisovelluksen? Sovellusta sinällään kiinnostaa ainoastaan access token jonka se saa vaihdettua haluamakseen tiedoksi eli autorisoinnista on kyse mutta eikö se jokaisen sovelluksen kohdalla ole samalla tavalla? Miten asia muuttuu siinä, että sovellus saa käyttöönsä tiedon siitä kuka olet? Onko oikeassa elämässä esimerkiksi pankkikortilla rahan nostaminen automaatista autorisointia vai autentikointia? Pankkikorttihan on kuin access token, jolla automaatti antaa käyttäjän tililtä rahaa - sikäli kun sinne käyttöoikeus kortilla on. Ei automaatilla kuitenkaan ole mitään käsitystä siitä, kuka sitä korttia oikeasti käyttää. Itselleni jäi vielä aika epäselväksi näiden kahden asian erot, vaikka niitä monessa paikassa esitelläänkin hyvin erilaisina asioina.

Json. Vaikka nyt osaankin luoda Javalla neljä sisäkkäistä luokkaa yhteen ja purkaa Jacksonilla JSON:a tehtyyn Java-olioon en ymmärrä miksi rakenteista pitää tehdä niin syviä. Ehkä tulevaisuus opettaa tietokantamalleista enemmän, mutta juuri nyt on vaikea ymmärtää miksi kolmannen tason listassa pitää olla vielä kaksi upotettua listaa lisää. Onko lähdetty yksinkertaisesta liikkeelle ja nälkä on kasvanut syödessä, vai onko tämä alun perinkin jo suunniteltu näin monitasoiseksi? Näin loppuvaiheessa työn tekemistä vielä huomio siitä, että Spotify tarjoaa nimenomaan kappaleiden hakuun parametrin *fields*, joka itseltä oli jäänyt sovellusta tehdessä huomaamatta. Ilmeisesti jossain muuallakin on herätty siihen, että dataa tulee kerralla liikaa. Tällä parametrilla voidaan rajoittaa JSON-muodossa tulevien kenttien määrää haluttuihin, eli toisin sanoen voidaan pyytää ainoastaan artistit ja kappaleen nimi kuten olisin halunnut tehdäkin. En kokeillut käytännössä tuon parametrin toimintaa ja sitä, että jättääkö se vielä rakenteen tasot jäljelle vai tuleeko sieltä yksitasoinen objekti paluupostissa takaisin. Ehkä siinä voisi olla myös hyvä optimointikohde sovelluksen jatkokehitykseen.

Spotify-tili linkittää sallitut sovellukset listalle, jossa ne pysyvät aina niin kauan kuin niiden pääsy tilin tietoihin evätään käyttäjän toimesta. Hyväksytty scope pysyy samana, vaikka sovellus ei enää esimerkiksi yksityisiä soittolistoja haluaisi tarkastellakaan. Ongelma tuli itselle vastaan tätä sovellusta tehdessä siinä, että olin hyväksynyt aluksi scopeksi yksityiset soittolistat. Tämä kuitenkin palautti jokaisella hakukerralla kehityskäyttöön liikaa soittolistoja ja yksittäisiä kappaleita, joten päätin muuttaa scopen koskemaan ainoastaan julkisia listoja. Lisäksi tein omalle Spotify-tililleni kaksi julkista listaa, joissa oli pieni valikoima sekä hittejä että kappaleita, joita ei hittitietokannasta löydy. Tässä vaiheessa Spotify kuitenkin palautti edelleen jokaisella kerralla sovellukselle myös tilini yksityiset soittolistat, koska en ollut tiliasetuksista peruuttanut Spot-A-Hit:lle myönnettyä lupaa tietoihini. Spotify onkin vasta syksyllä 2016 lisännyt ominaisuuden, jolla voi perua kolmannen osapuolen sovellusten pääsyn tilin tietoihin. Tämän ominaisuuden tulisi omasta mielestäni olla toiminnassa heti, kun OAuth 2.0 -protokollaa aletaan käyttää. Huonossa tapauksessa voi antaa vahingossa väärälle sovellukselle oikeuden kuunnella tai muokata soittolistoja kuitenkaan ilman mahdollisuutta perua oikeuksia. Voidaan siis ajatella, että yhtä tärkeää on pystyä kieltämään, kuin sallimaan, sovellusten pääsy omiin tietoihin.

Opinnäytetyötä kirjoittaessa opin paljon uusia asioita, joista valmiiseen versioon asti niistä ei päässyt kuin pieni osa. Itselleni jäi kuitenkin sellainen tuntuma, että nyt on hyvä käsitys siitä, miten OAuth 2.0 -protokollaa käytännön tasolla toteutetaan ja miten sitä käytetään eri palveluntarjoajien REST-rajapintojen kanssa. Työn johdannossa asetetut tavoitteet tulivat omasta mielestäni täytettyä ja sikäli kun työharjoittelupaikassa tuleva työnkuva ei muutu, niin uskoisin saaneeni ihan hyvät lähtökohdat tietojen hakemiseen REST-rajapinnoista käyttäen Javaa.

7.1 Google Playground

Tämän opinnäytetyön kirjoittamisen aikaan Google pitää yllä palvelua nimeltään Google OAuth 2.0 Playground. Siellä on mahdollista kokeilla omilla Google-tunnuksilla, miten Googlen toteutus OAuth 2.0 -kättelystä käytännössä tapahtuu. Mahdollista on muun muassa nähdä kokonaan kaikki lähetetyt ja vastaanotetut http-viestit. Tämä voi merkittävästi auttaa ymmärtämään OAuth 2.0 -mallia, kun näkee käytännössä http-viesteissä olevien header- ja body-osioiden sisällöt. Seuraavaksi kaksi kuvaa Googlen prosessista authorization coden vaihtamisesta access tokeniin. Googlen ja Spotifyn välillä on pieniä eroja parametritasolla, mutta periaate OAuth 2.0 -protokollan toteuttamisessa pysyy samana. Jos osaa tehdä Spotifyn kanssa pyynnin, niin myös Googlen palvelun käyttö onnistuu.

```
Request

POST /oauth2/v4/token HTTP/1.1
Host: www.googleapis.com
Content-length: 233
content-type: application/x-www-form-urlencoded
user-agent: google-oauth-playground

code=4%2FRAHfdsafDDWEofdpDNYm5W_u7P6k5vDyRy_1a4RV71qtmONhE&redirect_uri=https%3A%2F%2Fdevelopers.google.com
%2Foauthplayground&client_id=40543518192.apps.googleusercontent.com&client_secret=**********&scope=&grant_type=authorization_code
```

Kuva 21. Google Playgroundista otettu http-POST -viesti access tokenin pyyntöön. Erona suositukseen ja Spotifyn tapaan on, että client secret kulkee bodyssa parametrina eikä headerissa authorization-kentässä.

```
Response

HTTP/1.1 200 OK
Content-length: 266
X-xss-protection: 1; mode=block
X-content-type-options: nosniff
Transfer-encoding: chunked
Expires: Mon, 01 Jan 1990 00:00:00 GMT
Vary: Origin, X-Origin
Server: GSE
-content-encoding: gzip
Pragma: no-cache
Cache-control: no-cache, no-store, max-age=0, must-revalidate
Date: Fri, 31 Mar 2017 09:48:38 GMT
X-frame-options: SAMEORIGIN
Alt-svc: quic=":443"; ma=2592000; v="37,36,35"
Content-type: application/json; charset=UTF-8

{
  "access_token": "ya29.GlsfBHLasdfJJqM60cK2vtcs-h7USQvn39vj1D2sdxzRB3xgexRE27KAwRNssjD2193GS41UoSI32144x0j9s3wxSxyAwUojdWX_sKhhgZU1P2PFg3jl",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "1/O2OUxJQ12BGdRJUz3xXgUvT7zWEh21SuaUUle4t5zc4A"
}
```

Kuva 22. Google Playgroundista otettu http-POST-viesti access tokenin vastauksesta. Bodyssa tulee JSON-muotoinen vastaus jossa neljä parametria. Google ei oletusarvoisesti lähetä scope-parametria jos scope ei ole muuttunut. Sekä access-, että refresh tokenit ovat muokattuja, eivät alkuperäisiä.

7.2 **Demosovelluksen jatkokehitys**

Kuten aiemmin työn aikana on todettu, sovellus ei ole vielä valmis tuotantokäyttöön. En ole web-sovellusta ikinä julkaissut, joten varmuutta ei ole siitä, mitä kaikkia ominaisuuksia sovelluksessa tulee hyvän ohjelmointitavan mukaan olla sen lisäksi että ”se toimii”. Sovellukseen on tullut tässä opinnäytetyötä kirjoittaessa jo lisättyä muutamia pieniä parannuksia mitkä eivät loppukäyttäjälle näy, ja lisääkin olisi tarvetta tehdä. Seuraavaksi muutamia jatkokehitysideoita, joita on itselleni herännyt tässä opinnäytetyötä kirjoittaessa. Lisäsin myös suuntaa antavat aika-arviot kunkin idean toteutukseen.

7.2.1 Spring security

Nyt sovelluksen jokainen endpoint on suojaamaton eli kaikkialle pääsee kirjautumatta. Hieman aihetta sivuten lueskelin, että Spring Securityn olisi mahdollista käyttää suoraan OAuth 2.0 autorisointia kirjautumiseen eli sillä voitaisiin suoraan määritellä käyttäjätaso käyttäjälle (user, admin). Sovellusta ohjelmoidessa kokeilin osaisiko Spring security suoraan käyttää OAuth-kirjautumista autorisoinnin kanssa mutta se ei ainakaan itseltä onnistunut. Tämän saattamiseksi toimintaan kannattanee varata aikaa muutamasta tunnista kokonaiseen työpäivään riippuen siitä, minkälainen virallinen dokumentaatio on tai ovatko muut asian kanssa painineet kirjoittaneet siitä internetiin käytännönläheisiä artikkeleja.

7.2.2 OAuth 2.0 muuttujat

Nyt kaikki OAuth 2.0 client id:t ja client secretit, statet ja muut muuttuja ovat ainoastaan spotify.properties -tiedostossa. Tämä on kehityskäyttöön ihan hyvä ja nopea tapa koska niitä pitää muokata usein. Kuitenkin varsinaisessa tuotantokäytössä salaiset muuttujat kuten tietokannan nimet ja salasanat, samoin kuin muut salasanat pitäisi säilyttää turvallisesti verkossa vaanivien uhkatekijöiden varalta. Yksi mahdollisuus olisi tallettaa tiedot salattuna ympäristömuuttujiin, josta ainakin Spring osaa ne myös suoraan hakea. Spring hakee muuttujia automaattisesti 17 eri paikasta ja yksi vaihtoehto on sulkea muuttujat JSON-muotoon ja laittaa ne ympäristömuuttujaan tai järjestelmän ominaisuuteen (environment variable or system property). Tämä kuulostaa suhteellisen yksinkertaiselta muutokselta nykyiseen tilanteeseen, mutta siinä vaiheessa pitää olla jo aika varma siitä, ettei muuttujiin enää muutoksia tule koska niitä on isompi työ lähteä muuttamaan, jos tarvetta ilmenee. Jos mitään ei tarvitse muuttaa vaan ainoastaan vaihtaa muuttujien asuinpaikkaa niin aika-arvio ohjelmointiin pari tuntia.

7.2.3 State

State-parametrin merkitystä CSRF:n estämisessä painotetaan useassa paikassa. State-muuttujan tulisi olla istuntokohtainen, eli sen tulisi olla joka käyttäjällä erilainen, että siitä olisi jotain hyötyä. Tällä hetkellä sovelluksessa on kiinteästi yksi state-muuttuja, joka on tallennettu myös spotify.properties -tiedostoon. Varsinaisen suojauksen kannalta olisi ihan sama, onko tätä ollenkaan vai ei, mutta tekovaiheessa olikin tarkoitus vain tehdä parametrille paikka tulevaisuuden jatkokehitystä varten. Yksi mahdollinen tapa toteuttaa state-parametri on luoda satunnainen merkkijono käyttäjän cookiesta, kunhan se on palvelimella linkattuna käyttäjän istuntoon. Teoria on yksinkertaisen kuuloinen mutta koska en ole ikinä tällaista toteuttanut, antaisin varovaisen aika-arvion, joka on yksi työpäivä.

7.2.4 Virheentarkastelu

Virheentarkastelu on tekemättä lähes kokonaan. Sovellus ei ole varautunut mitenkään esimerkiksi siihen, että Spotifyn rajapinta palauttaa error-coden, vaan ainoa mitä tapahtuu on se, että selaimeen tulee http-statuskoodina esimerkiksi error 401 (unauthorized).

Ensimmäisessä vaiheessa todennäköisesti toteutettaisiin omat metodinsa ja verkkosivunsa parametreina tuleville error-koodeille, jolloin voidaan käyttäjälle suoraan kertoa mikä meni pieleen. Esimerkiksi access tokenin vanhenemista ei tarkastella millään tapaa. Hyvä tapa olisi lisätä yksinkertainen käyttäjätietokanta johon muun muassa access tokenin vanhenemisaika, sekä refresh token tallennetaan. Tällä voidaan vähentää turhia virheviestejä, kun access token uusitaan refresh tokenin avulla ennen ylimääräistä vanhentuneen access tokenin palauttamaa error-koodia. Ymmärtääkseni Spotify-login osaa kertoa käyttäjälle, jos

käyttäjätunnus tai salasana on mennyt väärin ja lisäksi sitä kautta voi jopa palauttaa unohdetun salasanan eli se itsessään ei vaadi mitään erillistä logiikkaa.

Seuraavassa vaiheessa olisi hyvä tehdä myös pari tarkastelua http-statuskoodeille. Javan response-entitystä voidaan suoraan katsoa millä statuksella vastaus tuli ja esimerkiksi yksinkertainen switch -logiikka tarkastelua varten olisi helppo toteuttaa. Springissä myös omien poikkeuksien (exception) lisääminen on helppoa yhdellä annotaatiolla, joten se voisi olla myös hyvä mahdollisuus saman koodin käyttämiselle muissakin controllereissa. Tämä helpottaisi tilannetta, jossa myöhemmässä vaiheessa sovellukseen haluttaisiin lisätä toimintoja.

Yleisestikin poikkeuksista puhuttaessa ne ovat käsittelemättä tässä sovelluksessa. Ainakin Jacksonin objectmapper-luokka voi heittää IOExceptionin, joten jokainen niistä pitäisi laittaa try-catch -osioihin ja jotenkin vielä oikeasti käsitelläkin mahdolliset tulevat poikkeukset.

Koska teoria ja tekniikka ovat jollain tapaa tuttuja, mutta työssä on monta pientä osaa, niin aika-arvio näiden toiminnallisuuksien tekemiseen on kaksi työpäivää.

8 YHTEENVETO

Tavoitteena oli luoda sovellus, joka OAuth 2.0 -protokollaa käyttäen tarkistaa sovelluksen käyttäjän Spotify-palvelun julkiset soittolistat ja vertaa niitä Suomen hitteihin vuosilta 2008-2015. Tavoitteessa onnistuttiin ja saatiin aikaan tässä kehitysvaiheessa julkaistavaksi soveltumaton sovellus, joka kuitenkin hoitaa vaaditut asiat OAuth 2.0 -protokollan mukaisesti.

Käytännön esimerkki toteutettiin Spotify API:n mukaisesti. Eroavaisuuksia viralliseen OAuth 2.0 -spesifikaatioon löytyi, mutta pääosin Spotify API noudattaa kuitenkin spesifikaation mukaista tapaa toimia. Nopealla tarkastelulla johdannossa mainittujen suurten palveluntarjoajien tarjoamissa rajapinnoissa käytännöt vaihtelevat aika vähän. Toiset toteuttavat virallista spesifikaatiota paremmin ja tiukemmin kuin toiset mutta periaate kaikissa on kuitenkin sama. Voidaan siis vahvistaa johdannon väite, että osatessaan käyttää Spotifyn rajapintaa onnistuu muidenkin palveluntarjoajien rajapintojen käyttö.

LÄHTEET

Boyd, Ryan. Getting Started With Oauth 2.0. Haettu 5.3.2017 osoitteesta http://profs.degroote.mcmaster.ca/ads/sartipi/courses/SC/w13/5.Resources/4.Security/Authorization/getting_started_with_oauth_2.0.pdf

D. Hardt, Ed. The OAuth 2.0 Authorization Framework (2012). Haettu 9.3.2017 osoitteesta https://tools.ietf.org/html/rfc6749

Google. Google OAuth 2.0 Playground. Haettu 13.3.2017 osoitteesta https://developers.google.com/oauthplayground/

Justin Richer. User Authentication with OAuth 2.0. Haettu 9.3.2017 osoitteesta https://oauth.net/articles/authentication/

OpenID. OpenID Connect Basic Client Implementer's Guide 1.0 - draft 37. Haettu 13.3.2017 osoitteesta http://openid.net/specs/openid-connect-basic-1_0.html

Spotify. Spotify developer. Haettu 10.3.2017 osoitteesta https://developer.spotify.com/web-api/

Spring boot reference guide. Haettu 17.3.2017 osoitteesta https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/